VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ZPRACOVÁNÍ SÍŤOVÉ KOMUNIKACE V DISTRIBUOVANÉM PROSTŘEDÍ

DISTRIBUTED NETWORK TRAFFIC PROCESSING

DIPLOMOVÁ PRÁCA
MASTER'S THESIS

AUTOR PRÁCE — **Bc. VILIAM LETAVAY**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JAN PLUSKAL**
SUPERVISOR

BRNO 2018

Zadání diplomové práce/20432/2017/xletav00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Letavay Viliam, Bc.**
Obor: Bezpečnost informačních technologií
Téma: **Zpracování síťové komunikace v distribuovaném prostředí**
Distributed Network Traffic Processing
Kategorie: Paralelní a distribuované výpočty

Pokyny:

1. Seznamte se s nástrojem Netfox Detective, obzvláště s moduly Netfox Framework zajišťující zpracování síťové komunikace.
2. Proveďte návrh řešení, které umožní distribuované zpracování síťové komunikace.
3. Implementujte s využitím .NET Core jako Docker containery a vše řádně testujte pomocí Unit Testů.
4. Proveďte výkonostní analýzu a srovnejte s jinými nástroji.

Literatura:

- Strauss, D. (2017). *C# 7 and .NET Core Cookbook*. Birmingham: Packt Publishing.
- Price, M. (2017). *C# 7 and .NET Core modern cross-platform development : create powerful cross-platform applications using C# 7, .NET Core, and Visual Studio 2017 or Visual Studio Code*. Birmingham, UK: Packt Publishing.
- MATOUŠEK, P., PLUSKAL, J., RYŠAVÝ, O., VESELÝ, V., KMEŤ, M., KARPÍŠEK, F. and VYMLÁTIL, M. (2015). *Advanced Techniques for Reconstruction of Incomplete Network Data*. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering., vol. 2015, no. 157, pp. 69-84. ISSN 1867-8211.

Při obhajobě semestrální části projektu je požadováno:

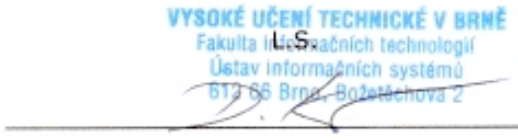
- body 1 a 2.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Pluskal Jan, Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018



L. S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Rozšírenie počítačových sietí a dostupnosti internetového pripojenia umožňuje našej spoločnosti rásť rýchlejšie ako kedykoľvek predtým. Zároveň tým ale otvára nové možnosti pre kybernetickú kriminalitu. Z tohto dôvodu vzniká u bezpečnostných administrátorov a vyšetrovacích zložiek potreba existencie nástrojov na analýzu zachytených dátových tokov. Táto diplomová práca sa venuje možnostiam analýzy zachytenej sieťovej komunikácie v distribuovanom prostredí, ktoré by umožnilo škálovať dostupné analyzačné schopnosti a prispôsobiť sa tým čoraz väčšiemu objemu dát prenesených po počítačovej sieti.

Abstract

Expansion of computer networks and availability of internet connection enables our society to grow faster then ever before. However, at the same time it opens up a new opportunuties for a cybercrime activities. That's why there is an increasing need of security administrators and law enforcing agencies for existence of a tools to analyze the captured data flows. This master thesis deals with ways of analysis of captured network traffic in a distributed environment, which would allow scaling of available analysis power and therefore adapt to ever increasing volumes of data transmitted over the computer networks.

Kľúčové slová

sieťová forenzná analýza, distribuované systémy, actor model, .NET Core, NTPAC

Keywords

network forensics, distributed systems, actor model, .NET Core, NTPAC

Citácia

LETAVAY, Viliam. *Zpracování síťové komunikace v distribuovaném prostředí.* Brno, 2018. Diplomová práca. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Pluskal

Zpracování síťové komunikace v distribuovaném prostředí

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Ing. Jana Pluskala. Uviedol som všetky literárne zdroje a publikácie z ktorých som čerpal.

.....
Viliam Letavay
24. mája 2018

Poďakovanie

Rád by som poďakoval vedúcemu diplomovej práce Ing. Janovi Pluskalovi za odborné vedenie, konzultácie, trpezlivosť a podnetné návrhy a rodine za podporu počas celého štúdia.

Obsah

Kapitola 1

Úvod

Sieťová forenzná analýza je vetva počítačovej forenznej analýzy, ktorú je možné definovať ako použitie vedecky preverených techník na zbieranie, zlúčenie, identifikáciu, preskúmanie, koreláciu, analýzu a dokumentovanie digitálnej evidencie z viacerých aktívnych a komunikujúcich digitálnych zdrojov pre účely odkrývania faktov spojených so zámerom, alebo s uskutočnením použitia neautorizovaných aktivít na narušenie, poškodenie, alebo kompromitovanie systémových komponentov a súčasné poskytnutie informácii nápomocných pri reakcii a zotavení z takýchto aktivít [7]. Ide teda o analýzu komunikácie medzi dvoma a viacerými zariadeniami komunikujúcimi v počítačovej sieti. Pomocou skúmania jednotlivých zachytených paketov rekonštruujeme udalosti tak, ako sa udiali v čase. Zaujímavé informácie, ktoré môžeme z tohto procesu získať, sú hlavne IP adresy koncových uzlov komunikácie, komunikačné protokoly a obsah vymieňaných protokolových (aplikačných) správ. Tieto informácie nazývame forenzné artefakty. Získané forenzné artefakty zrekonštruované zo zachytenej počítačovej komunikácie tak predstavujú cenný zdroj informácií pre bezpečnostných administrátorov a vyšetrovacie zložky pracujúce na objasnení bezpečnostných incidentov a kybernetickej trestnej činnosti.

Ciele práce

Cieľom tejto práce je návrh a implementácia distribuovaného systému určeného na vykonávanie sieťovej forenznej analýzy, stavajúc na základoch nástroja Netfox Detective. Vďaka distribuovanej architektúre vyšetrovateľ prestane byť limitovaný výkonom jediného počítača, ale bude schopný využiť výkon viacerých kooperujúcich počítačov zapojených do distribuovaného systému nazvanom NTPAC (Network Traffic Processing & Analysis Cluster). Od výsledného nástroja sa očakáva schopnosť rekonštrukcie L7 konverzácií z paketov zadaného PCAP súboru za pomoci automatického rozloženia výpočtu na viacero uzlov distribuovaného systému. Sekundárnym cieľom tejto práce je návrh a zostavenie modelu výpočtového clustera, na ktorom bude možné otestovať výslednú distribuovanú aplikáciu.

Obsah práce

Kapitola 2 popisuje problematiku rekonštrukcie zachytenej sieťovej komunikácie. V prvej časti popíše TCP/IP model, jeho jednotlivé vrstvy, radenie paketov do konverzácií na úrovniach týchto vrstiev a spôsob, akým môžeme reprezentovať prenesené aplikačné dáta. Následne bližšie popíše IP protokoly UDP a TCP, spôsoby akými prenášajú aplikačné dáta

a nakoniec ako je možné z týchto aplikačných dát rekonštruovať ďalej jednotlivé aplikačné správy.

V kapitole 3 sú popísané distribuované systémy spolu s modelmi súbežného vykonávania, konkrétne Actor model a Komunikujúce sekvenčné procesy. V druhej časti tejto kapitoly sa využijú tieto poznatky v architektonickom návrhu nástroja NTPAC za využitia Actor modelu, za ktorým nasleduje bližší návrh jeho podsystémov.

Po návrhu je v kapitole 4 popísaná implementácia jednotlivých komponentov navrhnutého systému v jazyku C# s využitím knižnice Akka.NET. V prvej časti je popísaná knižnica Akka.NET, implementácia Actor model systému. V ďalších dvoch častiach sú popísané implementácie uzlov *LoadBalancer* a *Reassembler*. V poslednej časti je popísané použitie výsledného nástroja a unit testovanie.

Kapitola 5 popisuje modelový cluster, ktorý bol zostavený na účely otestovania a predvedenia funkčnosti implementovanej distribuovanej aplikácie. Ďalej je v nej výkonnostné testovanie nástroja.

Kapitola 2

Problematika rekonštrukcie sieťovej komunikácie

Pri rekonštrukcii sieťovej komunikácie sa snažíme z jednotlivých zachytených paketov zrekonštruovať pôvodné aplikačné správy, ktoré boli vymenené medzi koncovými aplikáciami. Táto kapitola tak v podkapitole 2.1 popisuje model TCP/IP, vrstvy, na ktoré rozdeľuje komunikáciu a ako môžeme reprezentovať prenesené aplikačné dáta. Podkapitola 2.2 popisuje IP protokoly UDP a TCP a proces, pomocou ktorého môžeme z týchto protokolov rekonštruovať prenesené aplikačné dáta. Kapitola 2.3 nakoniec popisuje, ako z týchto dát môžeme získať aplikačné správy.

2.1 TCP/IP model

Sieťové zariadenia nekomunikujú medzi sebou výmenou sekvencií bitov zvolených konkrétnym výrobcom, ale dodržujú zavedené sieťové štandardy a protokoly. Vďaka tomu sú schopné vzájomnej komunikácie aj zariadenia rozdielnych výrobcov. V dnešnej dobe štandardne používaná sada protokolov TCP/IP definuje, ako by mali byť dáta rozdelené do paketov, adresované, prenášané, smerované a prijímané. Rozdeľuje komunikáciu na štyri vrstvy.

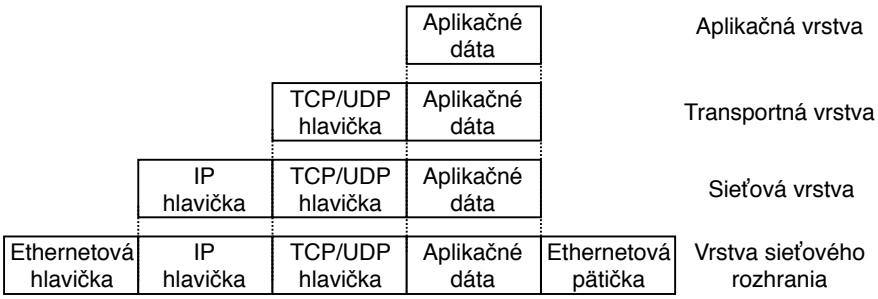
1. Vrstva sieťového rozhrania - L2. Definuje prístup k fyzickému médiu. Napríklad štandard IEEE 802.3 (známy pod názvom Ethernet) využívajúci MAC adresy.

2. Sieťová vrstva - L3. Zaručuje sieťovú adresáciu a smerovanie datagramov. Najpodstatnejšie protokoly pre naše potreby sú IPv4 a IPv6, ktoré využívajú IP adresy.

3. Transportná vrstva - L4. Využitá na odovzdávanie dát aplikáciám. Protokoly UDP a TCP využívajúce adresáciu pomocou čísiel portov. Protokol TCP navyše oproti protokolu UDP poskytuje spoľahlivé doručenie datagramov v správnom poradí a udržovanie spojení.

4. Aplikačná vrstva - L7. Obsahuje koncové aplikačné protokoly ako napríklad HTTP, DNS, FTP, ...

Protokoly danej vrstvy pridávajú svoju hlavičku obsahujúcu informácie spojené s týmto protokolom (napríklad zdrojová a cieľová IPv4 adresa IPv4 protokolu), ktorá je nasledovaná obsahom nižšej vrstvy. Pakety sú tak zostavené z dát aplikačného protokolu zapuzdreného



Obr. 2.1: TCP/IP model

v aplikačnom protokole, ďalej postupne zapuzdrenom v transportnom protokole, v sieťovom protokole a nakoniec v protokole sieťového rozhrania. Na obrázku 2.1 je zobrazená ukážka tohto zapuzdrenia (Ethernet pridáva popri svojej hlavičke aj pätičku na koniec dát obsahujúcu kontrolný súčet).

Pri analýze sieťovej komunikácie používame pojem konverzácia [9] – sekvencia paketov prenesených medzi dvoma koncovými bodmi. Podľa TCP/IP modelu rozlišujeme viacero úrovní granularity konverzácií, pričom konverzácie na nižšej vrstve rozdeľujú ďalej konverzácie na vyššej vrstve:

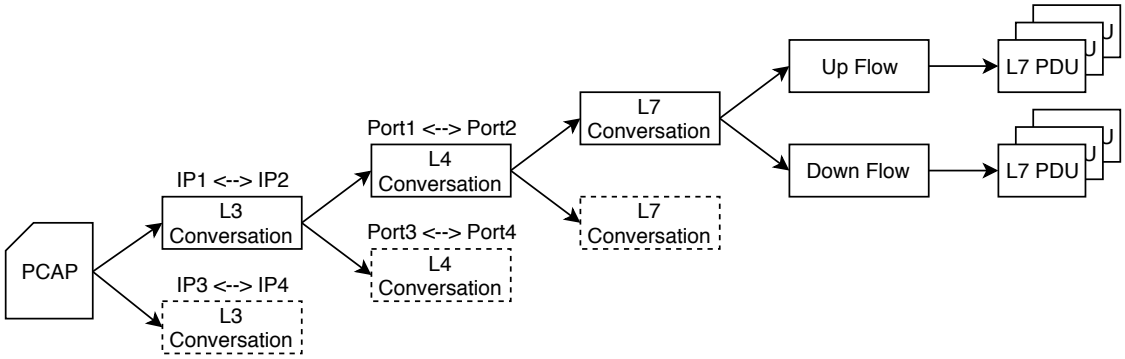
1. L3 konverzácie – pakety medzi medzi párom IP adries.
2. L4 konverzácie – pakety v jednej L3 konverzácii medzi párom portov a typom IP protokolu (TCP alebo UDP).
3. L7 konverzácie – pakety v jednej L4 konverzácii v jednom spojení transportného protokolu. L7 konverzácie (spojenia) si udržujú svoj stav a tak na rozdiel od rozlišovania L3 a L4 konverzácií sa nám už nestačí pozerať na pakety individuálne, ale musíme ich skúmať v kontexte ostatných paketov L4 konverzácie pomocou zložitejších heuristík. Tento proces je popísaný bližšie v podkapitole 2.2.

Podľa smeru paketov relatívne voči prvému paketu L7 konverzácie rozdeľujeme ďalej L7 konverzácie na *Up Flow*, pakety v relatívnom smere od iniciátora komunikácie a *Down Flow*, pakety v relatívnom smere k iniciátorovi spojenia. Zo sekvencie paketov v *Up Flow* a *Down Flow* následne pomocou procesu zvaného reassembling popísanom v podkapitole 2.2 extrahujeme dáta aplikačnej vrstvy rozdelené do tzv. L7 PDU. Každé z týchto L7 PDU následne obsahuje časovú známku, informáciu o smere (relatívne k iniciátorovi spojenia) a aplikačné dáta.

Na základe L7 PDU objektov a zo znalosti použitého aplikačného protokolu sme ďalej schopní zrekonštruovať obsah a význam prenášaných aplikačných dát. Obrázok 2.2 zhrňuje popísané rozdelenie paketov od L3 konverzácií až po L7 PDU.

2.2 Reassembling

Reassembling je proces rozpoznávania L7 konverzácií (spojení) v jednotlivých L4 konverzáciách a následnej extrakcie aplikačných dát – L7 PDU z týchto L7 konverzácií. Krok rozpoznania L7 konverzácií je nutný z dôvodu prípadného opätovného použitia portov pri komunikácii medzi dvoma zariadeniami. Iniciátor spojenia nie je nútený využiť vždy pre každé nové spojenie náhodný zdrojový port.



Obr. 2.2: Rozdelenie paketov

Protokol UDP je nespojovaný [10] a tak za UDP spojenie – L7 konverzáciu považujeme pakety UDP L4 konverzácie prenesené v jednom spoločnom časovom okne (s vhodnou toleranciou). Tento protokol nevyužíva žiadny mechanizmus na zaručenie spoľahlivosti doručenia dát v ich správnom poradí. Správy prenesené protokolom UDP sú vždy nositeľmi aplikačných dát[1]. Tieto správy považujeme za L7 PDU, doručené v poradí, v akom boli prenesené po sieti.

Na druhej strane, protokol TCP nadväzuje a aktívne udržiava spojenia a zaručuje spoľahlivý prenos dát v ich správnom poradí [11]. Na implementáciu týchto vlastností sú v hlavičke protokolu TCP zahrnuté sekvenčné a acknowledgement čísla[2] a sada kontrolných príznakov:

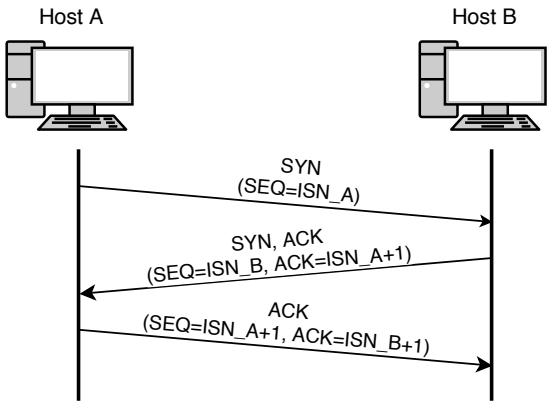
- SYN - odosielateľ inicializuje v smere k príjemcovi nové spojenie.
- ACK - odosielateľ signalizuje doručenie predošlej správy.
- FIN - odosielateľ dáva najavo protistrane, že nemá ďalšie dáta na odoslanie a ukončuje zo svojej strany spojenie.
- URG - odosielateľ oznamuje príjemcovi, že má spracovať túto správu prioritne.
- PSH - odosielateľ oznamuje príjemcovi, že má spracovať túto správu prioritne a odovzdať dáta priamo aplikácii bez toho, aby sa uložili do bufferu na neskoršie vyzdvihnutie.
- RST - nastavené pri odpovedi na nečakanú správu.
- ECE a CWR - využité pri detekcii zahltení spojenia.

TCP - Nadviazanie spojenia

Spojenia sú nadväzované pomocou tzv. Three-way handshake [11], procesom naznačeným na obrázku 2.3. Strana A, ktorá inicializuje spojenie, zašle strane B správu s príznakom SYN a náhodne vygenerovaným počiatočným sekvenčným číslom (Initial Sequence Number – ISN) ISN_A. Strana B odpovie správou s nastavenými príznakmi SYN a ACK, náhodne vygenerovaným počiatočným sekvenčným číslom ISN_B a acknowledgement nastaveným

[1] Na rozdiel od protokolu TCP, ktorý využíva navyše aj kontrolné správy určené na riadenie spojenia.
[2] Obe strany spojenia majú vlastné páry hodnôt týchto čísel.

na ISN_A + 1. Nadväzovanie spojenia je dokončené zaslaním správy od strany A, ktorá ma nastavený príznak ACK, sekvenčné číslo nastavené na ISN_A + 1 a acknowledgement na ISN_B + 1.



Obr. 2.3: TCP 3-way handshake

TCP - Prenos dát v ustanovenom spojení

Po nadviazaní spojenia môžu obe strany začať prenášať aplikačné dáta. Každá odoslaná správa má nastavené aktuálne sekvenčné a acknowledgement číslo na aktuálnu hodnotu odosielateľa [11]. Po odoslaní správy je hodnota odosielateľovho sekvenčného čísla inkrementovaná o dĺžku zaslaných dát. Po prijatí správy príjemca inkrementuje svoju hodnotu acknowledgement o dĺžku dát obsiahnutých v prijatej správe a odpovie pôvodnému odosielateľovi správou s nastaveným ACK príznakom, pomocou ktorej informuje odosielateľa o úspešnom doručení správy. Prijaté dáta príjemca neodovzdá okamžite cieľovej aplikácii, ale uloží ich do bufferu operačného systému. Uložené dáta sú odovzdané cieľovej aplikácii až v momente naplnenia bufferu alebo prijatia správy s nastaveným príznakom PSH.

Ošetrenie chybových stavov

Ak počas prenosu správy dôjde k poruche a následnému nedoručeniu správy, odosielateľ včas neprijme potvrdenie o doručení a opakuje pokus o doručenie opätovným zaslaním správy [11]. Protokol TCP umožňuje odosielateľovi odoslať viacero správ naraz bez čakania na potvrdenia doručení jednotlivých správ. Táto vlastnosť je využitá napríklad v prípade zasielania veľkého množstva aplikačných dát naraz, ktoré sú rozdelené na menšie celky a odoslané samostatne. V tomto prípade avšak môže dôjsť k preusporiadaniu jednotlivých správ, spôsobenému napríklad doručením paketov po viacerých rôzne rýchlych linkách. Príjemca je však schopný tieto pakety opätovne zoradiť do pôvodného poradia za použitia ich sekvenčných čísel.

TCP - Ukončenie nadviazaného spojenia

K ukončeniu odosielania z jednej strany a teda k polovičnému uzatvoreniu nadviazaného spojenia dôjde odoslaním správy s príznakom FIN, na ktorú odpovedá druhá strana správou s nastaveným príznakom ACK. Spojenie je uzatvorené úplne ukončením odosielania aj druhej strany [11].

TCP - Rekonštrukcia L7 PDU

Ako je možné vidieť, protokol TCP na rozdiel od protokolu UDP explicitne oddeľuje jednotlivé relácie komunikácie na transportnej vrstve. Vďaka tomu je reassemblovací algoritmus schopný jednoznačne rozpoznať jednotlivé L7 konverzácie v L4 konverzáciách. Ďalším rozdielom oproti protokolu UDP je, že protokol TCP pridáva výmenu správ, ktoré nie sú nositeľmi aplikačných dát (nadväzovanie spojenia, potvrdenie doručenia, uzatvorenie komunikačného kanálu, atď.) a tak neplatí, že každý paket L7 konverzácie predstavuje L7 PDU. Zároveň ani neplatí, že jedno L7 PDU pozostáva z jedného paketu, ale môže byť zložené z viacerých paketov doručených v nesprávnom poradí. Reassemblovací algoritmus tak musí byť schopný korektne (v správnom poradí, bez duplicít pri opätovnom prenose, atď.) rekonštruovať prenesené L7 PDU.

2.3 Rekonštrukcia aplikačných správ

Po rekonštrukcii L7 PDU z jednotlivých L7 konverzácií je možné začať analyzovať obsah prenesených aplikačných dát. Syntax a sémantika týchto dát zavisí od použitého aplikačného protokolu. Prvým krokom rekonštrukcie aplikačných správ L7 konverzácie je preto identifikácia použitého aplikačného protokolu. Na tú je možné použiť jednoduché rozpoznávanie na základe použitých čísel portov alebo zložitejšie heuristiky vykonávajúce inšpekciu obsahu aplikačných dát, ktoré ďalej využívajú štatistické metódy alebo metódy strojového učenia [6].

Na základe úspešne identifikovaného aplikačného protokolu môžeme zvoliť konkrétny algoritmus, ktorý je navrhnutý na rekonštrukciu správ daného aplikačného protokolu. V nasledujúcom kroku je zo zrekonštruovaných L7 PDU pomocou tohto algoritmu extrahovaná kolekcia vymenených aplikačných správ.

Postupné rozširovanie protokolu SSL/TLS a iných protokolov zabezpečujúcich šifrovanie komunikácie však tento postup výrazne sťažuje. Pri ich použití je obsah prenesených aplikačných dát zašifrovaný a za normálnych okolností ho nie je možné prečítať v dešifrovanej podobe, čo znemožňuje následnú rekonštrukciu aplikačných správ. Protokol SSL/TLS, založený na kryptografii verejného kľúča ale vieme dešifrovať za jednej z týchto okolností [2]:

1. Vlastníme privátny kľúč použitý pri nadväzovaní SSL/TLS relácie.

2. Máme možnosť využiť tzv. SSL/TLS Proxy, cez ktorú presmerujeme dátový tok medzi klientom a serverom a tak vykonáme tzv. man-in-the-middle útok, pri ktorom pred klientom predstierame, že sme pôvodný SSL server. Vďaka tomu sme schopní pri vytvorení SSL relácie použiť náš vygenerovaný privátny kľúč, ktorý môžeme neskôr použiť pri dešifrovaní. Nutnou podmienkou však je, že klient akceptuje nami predložený falošný certifikát vygenerovaného privátneho kľúča.

Kapitola 3

Návrh distribuovaného systému NTPAC

Pôvodným cieľom tejto práce bolo rozšírenie nástroja Netfox Framework o schopnosť distribuovanej analýzy zachytenej sieťovej komunikácie, čo by mu umožnilo škálovať podľa aktuálnej potreby jeho celkový výkon. Po hlbšom preskúmaní interného návrhu a fungovania nástroja Netfox Framework[8, 9] sa však ukázalo, že takéto riešenie by bolo príliš náročné na realizáciu z dôvodu problematickej integrácie s distribuovaným systémom a existujúcich chýb v návrhu a implementácii (neoptimálne spracovanie dát, nadmerné dátové závislosti, neškálovateľnosť). Táto práca sa preto bude ďalej venovať návrhu a implementácii nového nástroja NTPAC (Network Traffic Processing & Analysis Cluster), ktorý stavia na základoch nástroja Netfox Framework. Táto kapitola sa ďalej venuje jeho návrhu.

Podkapitoly 3.1 a 3.2 teoreticky popisujú distribuované systémy a modely súbežného vykonávania. Pochopenie týchto pojmov je kľúčové k nasledovnému návrhu fungovania celého nástroja. V podkapitole 3.3 je popísaný architektonický návrh distribuovaného systému nástroja NTPAC. Sú v ňom popísané hlavné fázy analýzy a uzly navrhnutého distribuovaného systému. Podkapitoly 3.4 a 3.5 bližšie popisujú návrh uzlov *LoadBalancer* a *Reassembler*.

3.1 Distribuované systémy

Distribuovaný systém je kolekcia kooperujúcich entít pracujúcich na dosiahnutí spoločného cieľa [5]. Jednotlivé entity sú autonómne a sú schopné komunikácie s ostatnými entitami distribuovaného systému pomocou počítačovej siete. Entita pozostáva z procesora a vlastnej pamäte. Distribuované systémy môžeme popísať týmito vlastnosťami [5]:

1. Absencia zdieľaných fyzických hodín. Udalosti v distribuovanom systéme nastávajú asynchrónne.
2. Absencia zdieľanej pamäte. Procesory navzájom komunikujú pomocou výmeny správ.
3. Geografické oddelenie. Entity distribuovaného systému nemusia byť nutne umiestnené v blízkej vzájomnej fyzickej vzdialenosti.
4. Autonómia a rôznorodosť. Jednotlivé entity môžu pre svoj beh využívať rôzny hardware a operačné systémy.

Distribuovaným programom nazývame množinu asynchrónnych procesov komunikujúcich prostredníctvom výmeny správ cez počítačovú sieť. Globálny stav distribuovaného vý-

počtu pozostáva zo stavov jednotlivých procesov a stavov ich komunikačných kanálov [5]. Podobnú architektúru predstavujú paralelné systémy. Tie sa od distribuovaných systémov líšia tým, že jednotlivé procesory sú súčasťou jedného fyzického celku (počítača) a navzájom komunikujú prostredníctvom zdieľanej pamäte.

3.2 Modely súbežného vykonávania

Pri návrhu a implementácii distribuovanej aplikácie je možné využiť viacero modelov popisujúcich súbežné vykonávanie. Dva najrozšírenejšie, využívajúce komunikáciu pomocou výmeny správ, sú Actor model a Komunikujúce sekvenčné procesy.

Actor model

Actor model je matematický model popisujúci súbežné vykonávanie programov predstavený Carlom Hewittom v roku 1973 [3]. Ďalej bol tento model popularizovaný jazykom Erlang úspešným použitím pri vytváraní vysoko paralelných a spoľahlivých telekomunikačných systémov. Za základnú jednotku vykonávania považuje actor, ktorý má svoj privátny stav (privátne premenné) a s druhými actormi interaguje výlučne prostredníctvom zasielania správ (nevyužíva zdieľanú pamäť). Actor ako reakciu na prijatú správu môže:

- odoslať konečný počet správ druhým actorom;
- vytvoriť konečný počet nových actorov (potomkov);
- zmeniť svoje správanie pri prijatí nasledujúcej správy.

Kolekcia actorov, ktorí sú schopní spolu komunikovať, sa nazýva actor systém. Zasielanie správ prebieha asynchrónne a teda operácia odoslania správy druhému actorovi je neblokujúca. Každý actor vlastní unikátnu adresu, ktorá je využitá pri špecifikovaní adresáta správy. Každý actor ďalej disponuje tzv. mailboxom, frontou do ktorej vkladá prijaté správy čakajúce na spracovanie. Keď je actor aktivovaný, vyberie prvú správu z fronty a spracuje ju príslušnou obslužnou rutinou (definovaným správaním) na základe typu prijatej správy. Plánovač zodpovedá za aktiváciu actorov po prijatí správ. Vďaka tomu, že actori nevyužívajú žiadne spoločné zdieľané prostriedky (zdieľané premenné) a komunikujú so sebou jedine pomocou správ, je možné aktivovať viacero actorov súbežne, bez toho aby sa ich vykonávania ovplyvnili. Jedinú podmienku, ktorú musí plánovač dodržať, je synchrónne vykonávanie jednotlivých actorov, tj. v jednom momente bude daný actor aktivovaný v maximálne jednom vlákne. Poradie aktivácie actorov nie je zaručené, ale poradie správ v mailboxe je dodržané. Jedným z efektov synchrónneho vykonávania actorov je, že pri implementácii súbežných programov postavených na Actor modeli nie sme nútení využívať synchronizačné prostriedky ako zámky a semafory.

Komunikujúce sekvenčné procesy (CSP)

Tony Hoare v roku 1978 definoval formálny model CSP [4], ktorý dnes využíva napríklad programovací jazyk Go[1]. V CSP je elementárnou jednotkou vykonávania proces. Jednotlivé procesy môžu medzi sebou komunikovať prostredníctvom výmeny správ vo vytvorených komunikačných kanáloch. Tie tak môžu byť využité na výmenu dát medzi procesmi a na

[1] The Go Programming Language - Why build concurrency on the ideas of CSP?, https://golang.org/doc/faq#csp

ich vzájomnú koordináciu. Proces môže do vytvoreného kanálu zaslať správu, prijať správu alebo uzatvoriť daný kanál. Komunikácia cez tieto kanály je synchrónna a proces, ktorý zasiela a zapisuje dáta do komunikačného kanála je zablokovaný do momentu, kedy si tieto dáta nevyzdvihne proces na jeho druhom konci.

Porovnanie

Actor model a CSP majú spoločné to, že oba využívajú na komunikáciu medzi jednotlivými jednotkami vykonávania, či už procesmi alebo actormi, výmenu správ. Líšia sa ale od seba v týchto vlastnostiach:

- CSP procesy sú anonymné, actori majú vlastnú identitu (adresu);
- CSP medzi sebou komunikujú synchrónne, actori asynchrónne;
- CSP využíva ustanovené komunikačné kanály na výmenu správ, actori špecifikujú pri zasielaní správy adresáta (adresu actora, ktorému má byť doručená daná správa).

Pre návrh a implementáciu nástroja NTPAC bol ďalej ako model súbežného vykonávania zvolený Actor model z dôvodu asynchrónnej komunikácie (actor môže zaslať asynchrónne správu a pokračovať ďalej vo vykonávaní) a možnej hierarchickej štruktúre actorov, ktorá by umožnila elegantne rozdeliť pakety do L3 až L7 konverzácií tak, ako boli popísané v kapitole 2.

3.3 Architektúrny návrh

Po popísaní distribuovaných systémov a modelov súbežného vykonávania je možné prejsť k architektúrnemu návrhu distribuovanej aplikácie. Celý proces rekonštrukcie zachytenej komunikácie je možné rozdeliť do dvoch základných fáz:

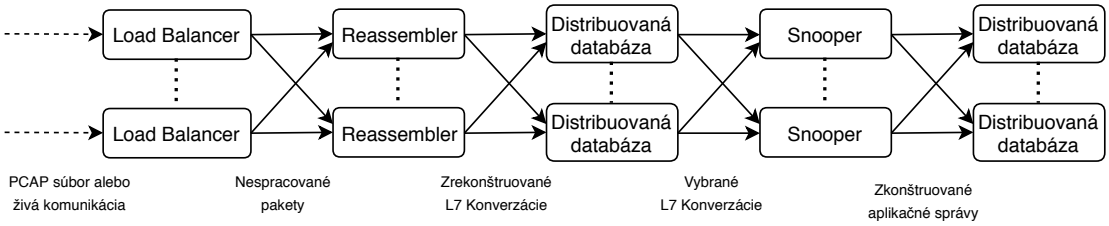
Predspracovanie dát Rekonštrukcia (reassembling) L7 konverzácií zo vstupnej zachytenej komunikácie. Každá z týchto konverzácií nesie primárne informácie o koncových uzloch komunikácie (IP adresy a porty), časové razítka a obsahy vymenených aplikačných dát.

Hĺbková analýza dát Identifikácia použitých aplikačných protokolov v zrekonštruovaných L7 konverzáciách a následné využitie vybraných modulov určených na rekonštrukciu správ konkrétneho aplikačného protokolu. Výstupom tejto fázy je kolekcia forenzných artefaktov ako napríklad navštívené HTTP stránky, prijaté a odoslané emaily alebo správy iného komunikačného protokolu – aplikácie, atď.

Na obidvoch fázach sa podieľajú množiny špecifických uzlov distribuovaného systému. Uzly, ktoré sa aktívne zúčastňujú procesu rekonštrukcie komunikácie[2], sú postavené na základe Actor modelu a tak všetky obsahujú hierarchiu lokálnych actorov participujúcich v jednom spoločnom actor systéme, schopných komunikácie s actormi ostatných uzlov. Každý uzol je vykonávaný na jednom stroji a na jednom stroji môže byť vykonávaných viacero uzlov.

Prvá fáza, **predspracovanie dát**, je vykonávaná na množine uzlov zvaných *Reassembler*. Tie samostatne (nezávisle od ostatných *Reassembler* uzlov) rekonštruujú L7 konverzácie z odovzdaného prúdu zachytených paketov, ktoré môžu pochádzať z PCAP súboru

[2]Nepatria sem teda napríklad databázové uzly.



Obr. 3.1: Architektúrny návrh nástroja NTPAC

alebo môžu byť aktívne zachytávané zo živého sieťového rozhrania. Zrekonštruované L7 konverzácie sú uložené do distribuovanej databázy. V podkapitole 3.5 je popísaný podrobnejší návrh týchto uzlov. V najčastejšom prípade však pracujeme len s jedným vstupným prúdom paketov (napríklad s jedným vstupným PCAP súborom), ktorý chceme spracovať. Preto, aby sme využili všetky inštancie zapojených *Reassembler* uzlov, musíme tento prúd rozdeliť na menšie podprúdy, ktoré rovnomerne rozdistribuujeme medzi aktívne *Reassembler* uzly. Vďaka tomu sme schopní dosiahnuť vyššiu využiteľnosť celého systému. Na toto rozdelenie využijeme ďalší typ uzlov – *LoadBalancer*. Tie prerozdeľujú pakety vstupného prúdu medzi aktívne *Reassembler* uzly podľa toho, do ktorej L4 konverzácie patria (jednotlivé pakety). Dôvod zvolenia práve tohto spôsobu rozdelenia a podrobnejší návrh uzlov *LoadBalancer* je popísaný v podkapitole 3.4.

V druhej fáze, **hĺbkovej analýze**, je podmnožina uložených zrekonštruovaných L7 konverzácií získaná z distribuovanej databázy a doručená na ďalší typ uzlov, tzv. *Snooper*. Tie identifikujú použitý aplikačný protokol L7 konverzácie a za použitia konkrétneho aplikačného dissectoru – snooperu, zrekonštruujú obsiahnuté aplikačné správy. Pri šifrovaní aplikačných dát v L7 konverzáciách protokolom SSL/TLS a znalosti použitého privátneho kľúča sú najskôr tieto dáta dešifrované. Extrahované informácie sú uložené späť do distribuovanej databázy. Táto práca sa ďalej venuje iba prvej fáze – rekonštrukcii L7 konverzácií. Hĺbková analýza zrekonštruovaných L7 konverzácií predstavuje jej možné pokračovanie.

Na diagrame 3.1 je graficky znázornený architektúrny návrh nástroja NTPAC. Zobrazuje jednotlivé typy uzlov a toky dát medzi nimi. Prepojenie medzi uzlami má logický charakter, je nezávislé od hardwardového zapojenia.

3.4 Návrh uzlu LoadBalancer

Ako bolo popísané v podkapitole 3.3, tento uzol prerozdeľuje vstupný tok paketov medzi aktívne inštancie *Reassembler* uzlov. Tieto pakety sú prijímané v nespracovanej podobe, sú vo forme bajtových polí tak, ako boli zachytené na sieti. Pred ich samotným prerozdelením je nutné ich spracovať, tzv . zparsovať až po vrstvu L4 a získať tak štrukturované informácie z jednotlivých hlavičiek použitých sieťových protokolov (po vrstvu L4). Po zparsovaní prichádza na rad samotné prerozdelenie medzi *Reassembler* uzly. Na to však nie je možné využiť naivnú metódu, ako napríklad Round Robin[3]. *Reassembler* uzly, ako bolo vysvetlené v podkapitole 3.3, rekonštruujú odovzdaný prúd paketov samostatne, bez komunikácie medzi sebou, ktorá by zaťažovala celý distribuovaný systém. A tak, aby boli schopné plne zrekonštruovať L7 konverzácie (ktoré môžu pozostávať z viacerých paketov), musia vlastniť všetky prípadné fragmenty konkrétnej L7 konverzácie. Použitím takejto naivnej metódy by

[3]Rozdelenie na základe cyklického prideľovania.

mohla nastať situácia, kedy by polovica paketov jednej L7 konverzácie skončila na jednom *Reassembler* uzle a druhá polovica na niektorom inom. Oba uzly by tak vlastnili len nekompletné dáta a ani jeden by nebol schopný plne zrekonštruovať danú L7 konverzáciu. Na rozdelenie vstupného toku je preto nutné využiť metódu, ktorá zachová kontext jednotlivých paketov, tj. pakety, ktoré spolu súvisia, musia byť spracovávané na práve jednom *Reassembler* uzle.

Tento kontext paketov predstavuje konverzácie, ako sme ich definovali v kapitole 2. Pri voľbe jemnosti rozpoznávania konverzácií máme na výber nasledovné možnosti:

L3 konverzácie Výpočtovo najjednoduchšie (pre každý paket stačí prečítať zdrojovú a cieľovú IP adresu), ale poskytuje najhrubšie rozpoznávanie, ktoré tak zvyšuje riziko nevyváženosti vyťaženia *Reassembler* uzlov.

L4 konverzácie Jemnejšie ako L3 konverzácie, no zároveň aj výpočtovo náročnejšie. Nutnosť defragmentácie možných IP fragmentov, aby sa získali kompletné dáta transportnej vrstvy fragmentovaných paketov.

L7 konverzácie Nedáva zmysel, pretože by sa vykonávala samotná práca *Reassembler* uzlov.

Pre rozlišovanie bola zvolená úroveň L4 konverzácií ako kompromis medzi výpočtovou náročnosťou a jemnosťou rozlišovania. Úlohou uzlu *LoadBalancer* je teda pre každý paket vstupného prúdu určiť jeho príslušnosť do danej L4 konverzácie a následne ho na základe tejto príslušnosti odovzdať vybranému *Reassembler* uzlu.

Príslušnosť paketu p do konverzácie sa získa zostavením tzv. kľúča konverzácie pomocou funkcie ConversationKey(p). Tá zostaví päticu (IP1, IP2, Port1, Port2, Protocol) obsahujúcu hodnoty:

- IP1 - menšia hodnota z dvojice zdrojovej a cieľovej IP adresy paketu p
- IP2 - väčšia hodnota z dvojice zdrojovej a cieľovej IP adresy paketu p
- Port1 - menšia hodnota z dvojice zdrojového a cieľového portu paketu p
- Port2 - väčšia hodnota z dvojice zdrojového a cieľového portu paketu p
- Protocol - transportný protokol paketu p (TCP alebo UDP)

Radenie IP adries a portov paketu podľa menších a väčších hodnôt zaručí, že pakety v oboch smeroch jednej L4 konverzácie budú mať rovnaký kľúč. Na priradenie jednotlivých L4 konverzácií konkrétnym *Reassembler* uzlom rozložíme priestor všetkých možných hodnôt kľúčov L4 konverzácií K do n[4] tried ekvivalencie binárnou reláciou R s využitím hashovacej funkcie Hash:

$$R = \{(x, y) \in K^2 \mid (Hash(a)\ mod\ n) = (Hash(b)\ mod\ n)\}$$

Nakoniec vytvoríme mapovanie z týchto tried ekvivalencií (číselných hodnôt) na aktívne *Reassembler* uzly. Na rozhodnutie o výbere *Reassembler* uzlu, na ktorom má byť spracovaný paket p, použijeme toto mapovanie, kde vyberieme konkrétny aktívny *Reassembler* uzol podľa triedy ekvivalencie c, ktorú vypočítame ako:

$$c = Hash(ConversationKey(p))\ mod\ n$$

[4]Hodnotu môžeme zvoliť napríklad ako 10-násobok počtu očakávaných *Reassembler* uzlov

Aby sme sa pri distribuovaní paketov jednotlivým *Reassembler* uzlom vyhli veľkej réžii spojenej so zasielaním a spracovaním veľkého počtu (relatívne) malých objektov, pakety nie sú zasielané individuálne. Pakety určené pre daný *Reassembler* uzol sú ukladané do zásobníka, ktorého obsah je po naplnení (na základe počtu paketov alebo ich akumulovanej veľkosti) odoslaný naraz vo forme dávky. Navrhnuté správanie implementuje actor PacketIngestorAndLoadBalancer, ktorý po prijatí požiadavky na zahájenie procesu rekonštrukcie špecifikovaného prúdu paketov začne s prijímaním a distribuovaním jeho jednotlivých paketov.

3.5 Návrh uzlu Reassembler

Úlohou tohto uzlu je prijímať dávky nespracovaných paketov z *LoadBalancer* uzlov, ich jednotlivé pakety znova zparsovať[5] a zo zparsovaných paketov zrekonštruovať obsiahnuté L7 konverzácie, ktoré sú ďalej uložené do perzistentného úložiska. Funkčné jadro tohto uzlu predstavuje reassemblovací mechanizmus. Na jeho podporu a prepojenie s distribuovaným systémom je navrhnutá postupne budovaná hierarchia actorov, ktorá je naznačená na obrázku 3.2. Úlohou týchto actorov je príjmať dávky nespracovaných paketov a rozdeliť ich do L4 konverzácií, ktoré tak môžu byť spracovávané reassemblovacím modulom separátne. Navrhnutá hierarchia pozostáva z týchto actorov:

Captures Controller Východiskový actor *Reassembler* uzlu. Prijíma dávky nespracovaných paketov od *LoadBalancer*-ov a pre každý z nich vytvára samostatného Capture actora. Prijaté dávky sú odovzdané príslušnému Capture actorovi. Vďaka tomu actorovi je možné, aby jeden *Reassembler* mohol prijímať dávky nespracovaných paketov od viacerých *LoadBalancerov* naraz.

Capture Predstavuje jeden zo vstupných prúdov nespracovaných paketov. Každý nespracovaný paket z odovzdanej dávky v prvom kroku zparsuje a následne podľa hlavičky sieťovej vrstvy určí L3 konverzáciu. Pre každú novú L3 konverzáciu je vytvorený nový L3 Conversation actor. V druhom kroku je daný paket odovzdaný príslušnému L3 Conversation actorovi.

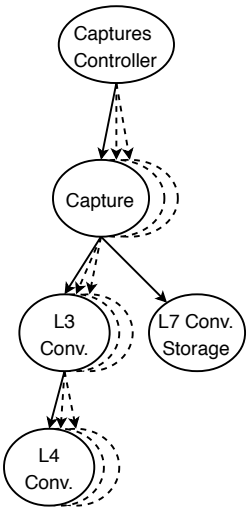
L3 Conversation Predstavuje L3 konverzáciu. Pre každý odovzdaný paket určí na základe hlavičky transportnej vrstvy L4 konverzáciu. Pre každú novú L4 konverzáciu je vytvorený nový L4 Conversation actor. Paket je následne odovzdaný príslušnému L4 Conversation actorovi.

L4 Conversation Predstavuje L4 konverzáciu. Každý odovzdaný paket je spracovaný reassemblovacím modulom. Zrekonštruované L7 konverzácie sú odovzdané L7 Conversation Storage actorovi na uloženie.

L7 Conversation Storage Odovzdané L7 konverzácie ukladá do perzistentného úložiska.

Po rozdelení paketov do L4 konverzácií je možné rekonštruovať z nich L7 konverzácie. Pre každú L4 konverzáciu je vytvorený samostatný reassemblovací modul so zvoleným reassemblovacím algoritmom určeným na rekonštrukciu daného transportného protokolu (TCP alebo UDP), ktorému sú postupne odovzdané pakety danej L4 konverzácie. Výstupom týchto modulov sú zrekonštruované L7 konverzácie.

[5] Prenášanie štrukturovaných informácií medzi uzlami by vyžadovalo ich serializáciu a deserializáciu, ktorá by bola príliš náročná na výpočtový výkon.



Obr. 3.2: Hierarchia Reassembler actorov

L7 konverzácia v sebe nesie informácie o koncových bodoch komunikácie (IP adresy, porty a transportný protokol), časové známky prvej a poslednej zaznamenanej aktivity (vymenených paketov) a dve kolekcie L7 PDU, pre každý smer komunikácie jednu. L7 PDU v sebe obsahuje časovú známku a samotný obsah zrekonštruovaných aplikačných dát.

Navrhnuté reassemblovacie algoritmy musia byť schopné rekonštruovať L7 konverzácie z transportných protokolov paketov L4 konverzácií, tak ako boli popísané v kapitole 2. Zároveň ich však musia zvládnuť rekonštruovať aj v prípadoch, kedy pracujú s neúplným zdrojom dát ako napríklad v prípade zachytenia len jedného smeru dátového toku, alebo pri strate jednotlivých paketov. Z tohto dôvodu UDP aj TCP Reassembler spracovávajú pakety L4 konverzácií v dvoch oddelených tokoch, v smere od iniciátora spojenia a v smere k nemu, z ktorých sú vytvorené L7 konverzácie.

UDP Reassembler

UDP Reassembler z každého odovzdaného paketu vytvorí samostatné L7 PDU (každý UDP paket predstavuje samostatné L7 PDU), ktoré vloží do jedného z dvoch tokov na základe smeru paketu. Pred týmto vložením však ešte vypočíta dobu medzi odoslaním tohto paketu a poslednou zaznamenanou aktivitou daného toku a ak je väčšia ako definovaná hodnota 10 minút, dané L7 PDU považuje za súčasť novej L7 konverzácie a tak z postupne budovaných tokov vytvorí jednu ukončenú L7 konverzáciu. Následne vyprázdni oba toky, do jedného z nich (na základe smeru paketu) vloží novovytvorené L7 PDU (ktoré spôsobilo ukončenie L7 konverzácie) a pokračuje ďalej v spracovávaní zvyšných paketov. Po spracovaní všetkých paketov L4 konverzácie je z oboch tokov vytvorená posledná L7 konverzácia.

TCP Reassembler

Požiadavka na zvládnutie rekonštrukcie aj neúplných dát je primárne relevantná pre TCP reassembler. Nemôže sa spoľahnúť na dáta z oboch smerov (napríklad pri analýze TCP handshaku), pretože ich nemusí mať vždy k dispozícii. Pakety L4 konverzácie tak musí spracovávať v dvoch separátnych tokoch, ktoré sa následne môže pokúsiť spárovať a tak

z nich vytvoriť jednu L7 konverzáciu. Ďalší problém predstavuje strata paketov. Musí teda počítať aj s prípadom ako napríklad stratené pakety TCP handshaku.

TCP Reassembler tak rovnako ako UDP Reassembler spracováva každý paket L4 konverzácie v samostatných jednosmerných tokoch. Tie však nevytvára priamo, ale pomocou reassemblerov tokov, ktoré sú vytvorené opäť dva, jeden pre každý smer. Reassembler tokov odovzdané pakety ukladá do svojej špeciálnej reassembling kolekcie.

Reassembling kolekcia zoraďuje odovzdané pakety podľa ich sekvenčných čísel. Pri detekcii pretečenia sekvenčných čísel paketov z dôvodu prirodzeného pretečenia 32 bitového čítača, alebo z dôvodu vytvorenia novej L7 TCP konverzácie, ktorá má nižšie ISN (Initial Sequence Number) ako predchádzajúca L7 TCP konverzácia, je zvýšené počítadlo pretečení sekvenčných čísel. K sekvenčným číslam použitým na radenie paketov je tak potom vždy pripočítaná hodnota 2^{32} vynásobená počtom pretečení. Touto normalizáciou sekvenčných čísel paketov dosiahne fakt, že sú jednak zoradené jednotlivé pakety v rámci L7 konverzácií, ale aj samotné L7 konverzácie sú zoradené za sebou v tomto priestore tak, nasledovali za sebou v čase. Paket s normalizovaným sekvenčným číslom s a veľkosťou L7 dát n tak vypĺňa interval <s, s + n> v priestore sekvenčných čísel. Vďaka tomu dokážeme ďalej detegovať opätovné zaslanie TCP správ (retransmission) pomocou zistenia zhody L7 dát u paketov, ktorých intervaly normalizovaných sekvenčných čísel sa prekrývajú.

Po spracovaní všetkých paketov L4 konverzácie týmto spôsobom oba reassemblery tokov postupne čítajú zoradené pakety zo svojich reassembling kolekcií a vytvárajú z nich L7 PDU. Pakety sú pridávané do jedného L7 PDU dokiaľ sa doňho nepridá paket s nastaveným TCP príznakom PSH (jedno L7 PDU tak môže byť tvorené viacerými paketmi). L7 PDU sú združené do samostatných jednosmerných L7 tokov pri detekcii nadväzovania alebo ukončenia TCP spojenia (paket s nastaveným TCP príznakom SYN alebo FIN). Pokiaľ bol zachytený TCP handshake jednosmerného L7 toku, hodnota ISN iniciátora spojenia sa nastaví ako identifikátor tohto toku. ISN iniciátora sa z pohľadu druhej strany vypočíta ako hodnota acknowledgment - 1 jej SYN paketu.

Po skončení tohto procesu TCP reassembler spáruje prislúchajúce vytvorené jednosmerné L7 toky na základe ich identifikátorov a z vytvorených dvojíc zhotoví finálne L7 konverzácie. Jednosmerné L7 toky bez identifikátorov (u ktorých nebol zachytený TCP handskake) sú spárované podla prieniku v čase, rovnako ako pri UDP L7 konverzáciách.

Kapitola 4

Implementácia nástroja NTPAC

Táto kapitola popisuje implementáciu nástroja NTPAC, navrhnutého v kapitole 3. Nástroj je implementovaný v jazyku C# s využitím frameworku .NET Core 2.0[1], multiplatformovou open-source implementáciou C# runtimu vyvíjanou spoločnosťou Microsoft. Vďaka tomu je výsledný softwarový produkt spustiteľný na systémoch Windows, macOS a Linux, na ktorý je primárne zameraný z dôvodu nasadenia na počítačovom clusteri. Je postavený na C# knižnici Akka.NET, poskytujúcej možnosti pre výstavbu paralelných distribuovaných systémov pri využití Actor modelu. Vďaka tomu sa mohol vývoj zamerať hlavne na implementáciu uzlov *LoadBalancer* a *Reassembler*, bez zbytočnej reimplementácie funkcionality distribuovaného systému ako takého, ako napríklad vzájomná koordinácia jednotlivých uzlov a zasielanie správ medzi nimi.

Prvá podkapitola 4.1 popisuje knižnicu Akka.NET, jej základné rysy a komponenty, ktoré boli využité pri implementácii nástroja NTPAC. Podkapitola 4.2 popisuje štrukúru projektu. Podkapitoly 4.3 a 4.4 popisujú samotnú implementáciu uzlov *LoadBalancer* a *Reassembler*. Posledná podkapitola 4.5 popisuje použitie výsledných spustiteľných súborov a testovanie implementácie.

4.1 Akka.NET

Akka.NET[2] je C# portom Java knižnice Akka, implementujúca Actor model. Je určená na výstavbu paralelných distribuovaných programov, ktorým umožňuje [1]:

1. Implementáciu viacvláknových programov bez nutnosti využívania synchronizačných prostriedkov ako zámky a semafory.
2. Transparentnú sieťovú komunikáciu medzi uzlami distribuovaného systému.
3. Návrh škálovateľných, vysoko dostupných distribuovaných systémov.

Actorov je možné vytvoriť implementovaním novej triedy odvodenej z nadtriedy `ReceiveActor`. Pomocou zdedenej metódy `Receive` je možné zaregistrovať obslužnú rutinu (metódu) pre daný typ prijatej správy. Správy sú inštanciami ľubovoľných tried a ich typy, ktoré sú využité pri voľbe obslužných rutín sú reprezentované ich konkrétnymi triedami. Ďalšou užitočnou zdedenou metódou je `Become`, pomocou ktorej je možné definovať aktuálne správanie actora zavolaním s odovzdanou metódou, ktorá v sebe pomocou metód

[1] .NET Core, https://github.com/dotnet/core
[2] Akka.NET, https://github.com/akkadotnet/akka.net

`Receive` nastaví reakcie na prijaté správy. Actori sú reprezentovaní rozhraním `IActorRef`[3], ktoré mimo iných obsahuje tieto metódy:

- `Tell(message)` - neblokujúco zašle správu actorovi, nečaká na odpoveď.
- `Ask(message)` - blokujúco zašle správu actorovi a čaká od neho odpoveď.
- `Forward(message)` - neblokujúco prepošle prijatú správu druhému actorovi so zachovaním informácie o pôvodnom odosielateľovi.
- `Path()` - získa adresu actora.

Pri spracovávaní prijatej správy obslužnou rutinou obsahuje zdedená vlastnosť `Sender` referenciu `IActorRef` odosielateľa správy. Na zasielané správy sa kladie požiadavka, aby boli jedenkrát vytvorené a ďalej už nemenné. To umožňuje bezpečné zaslanie jednej správy viacerým actorom, bez rizika vzniku race conditions.

Remoting

Akka.NET umožňuje zasielanie správ medzi actormi jedného actor systému jednak lokálne, v rámci jedného procesu, ale aj vzdialene naprieč rozdielnymi procesmi, na iných počítačoch. Každý actor má pridelenú vlastnú logickú adresu, pomocou ktorej mu môžu ostatní actori v actor systéme zasielať správy. Zasielanie správ je transparentné vzhľadom k lokácii a tak programátor pri implementovaní vlastnej funkcionality nemusí rozlišovať medzi lokálnym a vzdialeným zasielaním. Knižnica Akka.NET taktiež umožňuje aj vzdialené vytváranie actorov, tj. actor v jednom procese môže vytvoriť nového actora v druhom procese (na druhom počítači). V základnom nastavení je na sieťovú komunikáciu medzi dvoma procesmi využitý TCP kanál, implementovaný knižnicou DotNetty[4].

Cluster

Vzdialené zasielanie správ je posunuté o krok ďalej s podporou združovania jednotlivých Akka.NET procesov (počítačov) do väčších jednotných celkov – clusterov. Akka.NET cluster predstavuje škálovateľnú, decentralizovanú peer-to-peer sieť Akka.NET procesov bez existencie jediného bodu zlyhania. Akka.NET umožňuje jednotlivým účastníkom clustera objavovať nových účastníkov a detegovať odpojenie existujúcich. Keď sa chce uzol pripojiť do clustera, kontaktuje najprv jeden z jeho nakonfigurovaných seed uzlov, ktorých adresy sú známe každému uzlu. Po pripojení pripájajúci sa uzol obdrží od seed uzlu informácie o všetkých pripojených účastníkoch clustera. Každý uzol má pridelenú množinu rolí, ktoré môže v clusteri plniť. Pomocou nich sú ostatné uzly schopné vyhľadávať konkrétne typy pripojených uzlov, ktoré plnia danú funkciu v clusteri.

Ďalšou funkcionalitou Akka.NET clustrov je tzv. sharding, ktorá umožňuje zasielanie správ pomocou kľúčov týchto správ. Hodnoty týchto kľúčov určujú actora v clusteri, tzv. entitu, ktorá túto spravu príjme a spracuje. Priestor hodnôt kľúčov je rozdelený medzi entity, ktoré sú rozložené na uzloch clustera. Uzly, na ktorých sa majú vytvárať entity, je možné bližšie špecifikovať danou rolou. Pri používaní sú tak na vybraných uzloch automaticky vytvárané sharding entity podľa potreby. Akka.NET sa snaží udržiavať počet

[3] Actori tak nevlastnia priame odkazy na inštancie druhých actorov, ale len špeciálne referencie na nich v actor systéme.

[4] DotNetty, https://github.com/Azure/DotNetty

vytvorených sharding entít na jednotlivých uzloch v rovnováhe. Tento mechanizmus je tak vhodný pri implementácii distribuovania práce medzi viacero uzlov clustera. Pri samotnom implementovaní tejto funkcionality tak len vytvoríme špeciálneho proxy actora, ktorému následne môžeme zasielať správy, ktoré chceme doručiť na jednu z entít. Tieto správy pred ich odoslaním obalíme v špeciálnej obálke obsahujúcej okrem samotnej správy aj kľúč, ktorý sa použije pri výbere cieľovej entity. Akka.NET následne zaistí doručenie správy na jednu z entít a jej prípadné vytvorenie na niektorom z uzlov. Pri vytváraní sa snaží udržať počet vytvorených entít na uzloch v rovnováhe.

Streamy

Poslednou pre nás podstatnou funkciou, ktorú Akka.NET poskytuje, sú tzv. streamy. Tie umožňujú spracovávania dát na vyššej úrovni abstrakcie pomocou zreťazeného spracovávania dát v oddelených blokoch. Každý zapojený blok tak vstupné dáta transformuje na výstupné dáta, ktoré vstupujú ďalej do nasledujúceho bloku. Dáta tak tečú od pôvodného zdroja dát zvaného *Source*, ku konečnému cieľu zvanému *Sink*. Akka.NET streamy plne implementujú Reactive Streams[5] špecifikáciu.

4.2 Štruktúra projektu

Z dôvodu väčšej prehľadnosti zdrojových súborov a jednoduchšej rozšíriteľnosti je implementácia NTPAC-u rozdelená do viacerých oddelených projektov, ktoré sú združené do jedného C# Solution. To obsahuje nasledovné projekty:

1. NTPAC.Common – spoločné triedy využívané ostatnými projektami. Tento projekt obsahuje v oddelených priečinkoch rozhrania, modely, rozšírenia, továrne a enumy.
2. NTPAC.Actors – implementácia actorov.
3. NTPAC.Messages – správy vymieňané medzi actormi.
4. NTPAC.PcapLoader – implementácia modulu určeného na čítanie zachytených paketov z PCAP súborov.
5. NTPAC.LoadBalancer – implementácia LoadBalancer uzlu.
6. NTPAC.LoadBalancerCli – spustiteľný program, rozhranie k NTPAC.LoadBalancer.
7. NTPAC.Reassembler – spustiteľný program, implementácia Reassembler uzlu.
8. NTPAC.Reassembling – implementácia TCP a UDP reassemblovacích modulov.
9. Lighthouse.NetCoreApp – spustiteľný program, seed uzol potrebný pre zostavenie Akka.NET clustra[6].

[5] Reactive Streams, https://www.reactive-streams.org
[6] Lighthouse, https://github.com/petabridge/lighthouse

Modely

Modely sú triedy, ktorých hlavnou úlohou je byť nositeľom dát. Spoločné modely sa nachádzajú sa v projekte NTPAC.Common, v namespace NTPAC.Common.Models. Najdôležitejšie z nich sú:

`Capture` Reprezentuje zdroj nespracovaných paketov. Informácie o ňom sú obsiahnuté v objekte triedy `CaptureInfo`.

`CaptureInfo` Obsahuje informácie o zdrojovom PCAP súbore (názov súboru).

`Frame` Reprezentuje zparsovaný paket. Implementuje rozhranie `IFrame`, ktorého triedny diagram je na obrázku 4.1. Obsahuje časovú známku, informácie o koncových bodoch (IP adresy a porty), použitý transportný protokol, informácie vyextrahované z TCP hlavičky (príznaky, sekvenčné číslo, acknowledgment číslo) a samotné aplikačné dáta v podobe bajtového poľa.

`L3ConversationKey` L3 kľúč využitý pri rozhodovaní o príslušnosti paketu do L3 konverzácie. Je zostavený zo zdrojovej a cieľovej IP adresy paketu.

`L4ConversationKey` L4 kľúč využitý pri rozhodovaní o príslušnosti paketu do L4 konverzácie. Je zostavený zo zdrojového a cieľového portu a transportného protokolu paketu.

`L3L4ConversationKey` Združuje `L3ConversationKey` a `L4ConversationKey` do jedného objektu.

`L7Pdu` Reprezentácia zrekonštruovaného L7 PDU. Implementuje rozhranie `IL7Pdu`, ktorého triedny diagram je na obrázku 4.1. Obsahuje časové známky, smer a aplikačné dáta.

`L7Conversation` Reprezentácia zrekonštruovanej L7 konverzácie. Implementuje rozhranie `IL7Conversation`, ktorého triedny diagram je na obrázku 4.1. Obsahuje informácie o koncových bodoch komunikácie a použitom transportnom protokole, časové známky a kolekciu zrekonštruovaných L7 PDU v podobe `L7Pdu` objektov. Z tejto kolekcie obojsmerných L7 PDU ďalej umožňuje získať jej filtrovaním aj kolekcie L7 PDU v jednotlivých smeroch (*Up* a *Down*).

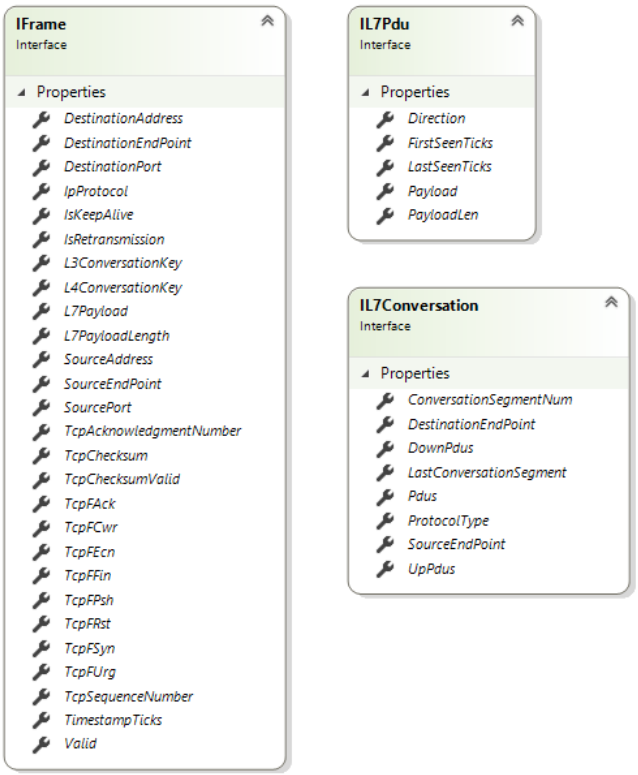
Triedy `Capture` a `L7Conversation` implementujú ďalej rozhranie `IDataEntity`. Triedy implementujúce toto rozhranie sú určené na perzistentné uloženie (obsahujú atribút `Id`, obdobu primárneho kľúča relačnej databázy).

Správy vymieňané medzi actormi

Jednotlivé správy, ktoré sú zasielané a prijímané actormi sú inštanciami na toto určených tried. Tieto triedy sa nachádzajú v projekte v NTPAC.Messages, v rovnomennom namespace. Najdôležitejšie z nich sú:

`StartProcessingRequest` Požiadavka pre actora `PacketIngestorAndLoadBalancer` na zahájenie distribuovania paketov zo vstupného zdroja. Obsahuje inštanciu rozhrania `IPcapLoader`, ktorá je využitá na čítanie jednotlivých nespracovaných paketov zo vstupného zdroja a inštanciu objektu `CaptureInfo` bližšie popisujúcu tento vstupný zdroj paketov.

`ProcessingResult` Informovanie iniciátora analýzy o jej ukončení spolu s dodatočnými informáciami ako celkový čas rekonštrukcie, počet spracovaných paketov, atď.



Obr. 4.1: Triedne diagramy rozhraní `IFrame`, `IL7Pdu` a `IL7Conversation`

`CaptureInfoRequest` Vyžiadanie si informácií o aktuálne spracovávanom zdroji nespracovaných paketov od actora `PacketIngestorAndLoadBalancer`.

`ProcessRawPacketRequest` Požiadavka na spracovanie zachyteného nespracovaného paketu.

`ProcessRawPacketBatchReponse` Dávka požiadaviek na spracovanie zachytených nespracovaných paketov.

4.3 Implementácia uzlu LoadBalancer

Jadro tohto uzlu tvorí actor `PacketIngestorAndLoadBalancer`, ktorého triedny diagram je na obrázku A.1. Po vytvorení čaká na príjem správy typu `StartProcessingRequest`, ktorá je obslúžená metódou `OnStartProcessingRequest`. Tá uloží referenciu odosielateľa tejto správy (aby po skončení rekonštrukcie mohol byť informovaný o jej výsledku) do inštančnej premennej `Contractor`, uloží hodnoty `PcapLoader` a `CaptureInfo` prijatej požiadavky, metódou `ProcessingCaptureBehaviour` použitou ako parametrom metódy `Become` zmení svoje správanie na príjem správ typu `CaptureInfoRequest`, `ProcessRawPacketBatchReponse` a `ProcessingResult` a nakoniec zavolá asynchrónnu metódu `StartProcessingAsync`, ktorou spustí proces rekonštrukcie vstupných paketov.

Metóda `StartProcessingAsync` v prvom kroku vytvorí sharding proxy actora, ktorého entity budú typu `CapturesControllerActor`. Títo actori sa tak budú vytvárať automaticky na *Reassembler* uzloch a budú im automaticky smerované správy zaslané vytvorenému sharding proxy actorovi.

V ďalšom kroku zostaví Akka.NET stream, ktorého úlohou je čítať vstupné nespracované pakety z odovzdaného `PcapLoader` objektu a distribuovať ich po dávkach *Reassembler* uzlom. Zdroj dát *Source* otvorí a postupne číta jednotlivé nespracované pakety typu `RawCapture` z odovzdaného objektu `PcapLoader` typu `IPcapLoader`. Rozhranie `IPcapLoader` definuje všeobecný zdroj nespracovaných paketov a konkrétna implementácia ich tak môže čítať z ľubovolného zdroja, či už z PCAP súboru alebo zo živého sieťového rozhrania.

Pipeline

Jednotlivé elementy zo zdroja (nespracované pakety) sú odovzdané do pipeline vytvorenou metódou `CreatePipeline`. Jej jednotlivé bloky a toky dát medzi nimi sú zobrazené na diagrame 4.2. Vytvorená pipeline v prvom bloku transformuje každý nespracovaný paket metódou `ParsePacket` na nový objekt typu `ProcessRawPacketRequest`. Ten obsahuje hodnoty daného nespracovaného paketu (časovú známku, typ linky, na ktorej bol paket zachytený a dáta paketu vo forme bajtového poľa) a hodnotu `EntityId`, ktorá je využitá pri neskoršom smerovaní na konkrétnu entitu sharding mechanizmom. `EntityId` je vypočítaná z hashu objektu `L3L4ConversationKey` modulo počet entít. Pre jeho vytvorenie je nutné daný nespracovaný paket zparsovať, aby bolo možné vyčítať informácie z jeho L3 a L4 hlavičiek. Na parsovanie paketov je využitá knižnica Packet.Net[7]. Metóda `ParsePacket` taktiež aj defragmentuje fragmentované IP pakety, aby bolo možné prečítať ich L4 hlavičky.

V nasledujúcom bloku pipeline je prúd `ProcessRawPacketRequest` objektov rozdelený podľa ich hodnôt `EntityId` do samostatných podprúdov. V týchto podprúdoch sú blokom `GroupedByCountAndRawCaptureSizeWithin` jednotlivé `ProcessRawPacketRequest` objekty združené do dávok o danom maximálnom počte objektov a o danej maximálnej veľkosti ich nespracovaných paketov. K dávkam `ProcessRawPacketRequest` objektov v podprúdoch je následne blokom `ZipWithIndex` priradené ich sekvenčné číslo určujúce ich poradie v danom podprúde. Podprúdy sú následne spojené späť do jedného prúdu dávok `ProcessRawPacketRequest` objektov.

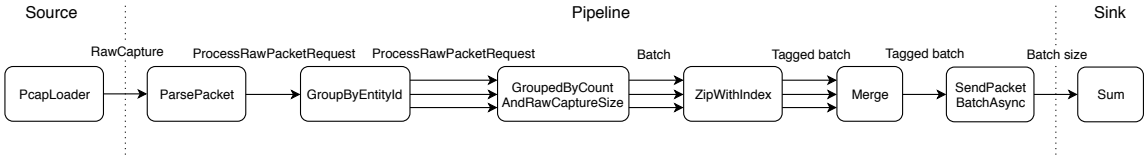
Jednotlivé dávky `ProcessRawPacketRequest` objektov zo zjednoteného prúdu sú nakoniec odoslané metódou `SendPacketBatchAsync` príslušným `CapturesControllerActor` entitám vytvorených na *Reassembler* uzloch. `SendPacketBatchAsync` zaobalí danú dávku do nového objektu typu `ProcessRawPacketBatchRequest`, ktorý ďalej zaobalí do sharding obálky, ktorú následne zašle sharding proxy actorovi, ktorý ju prepošle príslušnej `CapturesControllerActor` entite zvolenej na základe hodnoty `EntityId`[8]. Po odoslaní tejto obálky sa čaká na odpoveď typu `ProcessRawPacketBatchResponse`. V jednom momente je prenášaných viacero dávok súčasne. Z tohto bloku vystupujú počty rozdistribuovaných paketov v dávkach, ktoré sú následne akumulované na konci streamu, v *Sink*-u.

Finalizácia distribuovania

Po vyčerpaní vstupných nespracovaných paketov z objektu `PcapLoader` je uzatvorená pipeline a metódou `SendFinalizeProcessingRequestsToReassemblersAsync` je všetkým vytvoreným `CapturesControllerActor` entitám odoslaná požiadavka na dokončenie reassemblovania. Táto požiadavka je implementovaná `ProcessRawPacketBatchRequest` objektom s nastaveným príznakom `FinalizeCapture`.

[7] Packet.Net, https://github.com/chmorgan/packetnet

[8] Všetky `ProcessRawPacketBatchRequest` objekty v dávke majú túto hodnotu rovnakú.



Obr. 4.2: Diagram blokov LoadBalancer pipeline

V poslednom kroku je iniciátor analýzy informovaný o jej ukončení zaslaním správy `ProcessingResult`, ktorá obsahuje počet spracovaných paketov a celkový čas strávený ich analýzou. Po odoslaní tejto správy actor `PacketIngestorAndLoadBalancer` znova začne čakať na príjem novej požiadavky na zahájenie analýzy.

Spustiteľný program

Trieda `LoadBalancer` vytvára prepojenie medzi `PacketIngestorAndLoadBalancer` actorom a zvyšnou implementáciou mimo actor systému. Konštruktor tejto triedy preberá inštanciu rozhrania `IPcapLoader`, ktorá predstavuje zdroj nespracovaných paketov. Po vytvorení tohto objektu je možné zavolaním metódy `OpenPcapAsync` vytvoriť nový Akka.NET actor systém, v ňom vytvoriť nového `PacketIngestorAndLoadBalancer` actora a inštruovať ho zaslaním správy `StartProcessingRequest` aby začal spracovávať vstupný zdroj nespracovaných paketov. Následne sa počká na príjem správy `ProcessingResult`, indikujúcej ukončenie spracovávania.

Objekt `LoadBalancer` je vytvorený inštanciou triedy `LoadBalancerCli`, ktorá obsahuje vstupný bod spustiteľného programu NTPAC.LoadBalancerCli. V tejto triede dochádza aj k vytvoreniu inštancie rozhrania `IPcapLoader`, objektu triedy `PcapLoader`. `PcapLoader` implementuje čítanie nespracovaných paketov z PCAP súborov pomocou knižnice SharpPcap[9], ktorá je C# wrapperom pre knižnicu libpcap[10]

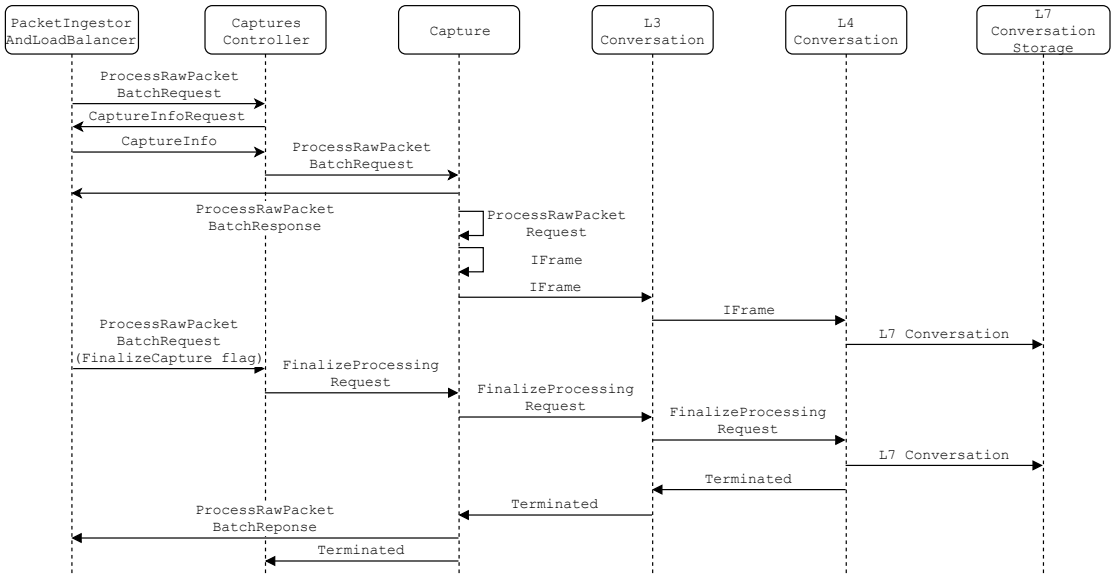
4.4 Implementácia uzlu Reassembler

Funkcionalita *Reassembler* uzlu je implementovaná na základe návrhu v podkapitole 3.5. Pozostáva z hierarchie actorov, ktorí prijímajú dávky nespracovaných paketov od actora `PacketIngestorAndLoadBalancer`, rozdeľujú jednotlivé pakety do L4 konverzácií a následne z nich reassemblujú L7 konverzácie, ktoré uložia do perzistentného úložiska. Koreňový projekt tohto uzla je NTPAC.Reassembler, v ktorom sa nachádza vstupný bod spustiteľného programu NTPAC.Reassembler.

4.4.1 Implementácia actorov

Sekvenčný diagram na obrázku 4.3 popisuje zasielané správy medzi `PacketIngestorAndLoadBalancer` a actormi *Reassembler* uzlu. Hierarchické usporiadanie týchto actorov bolo zobrazené na obrázku 3.2.

[9]SharpPcap, https://github.com/chmorgan/sharppcap
[10]TCPDUMP/LIBPCAP public repository, http://www.tcpdump.org/

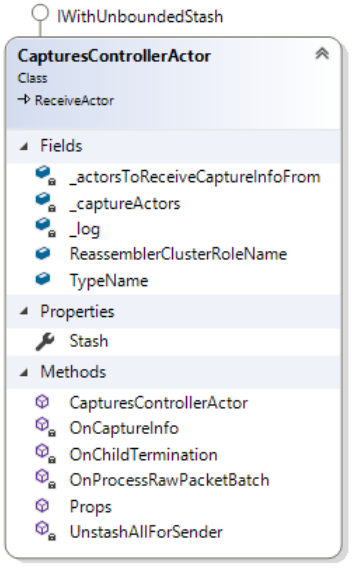


Obr. 4.3: Interakcia medzi actormi

Captures Controller

`CapturesController` actor (diagram na obrázku 4.4) je vytváraný automaticky ako sharding entita. Pre každého `PacketIngestorAndLoadBalancer` actora, ktorý mu zasiela dávky nespracovaných paketov `ProcessRawPacketBatchRequest`, vytvára samostatného `Capture` actora, ktorému ďalej preposiela tieto dávky.

Pomocou slovníka `captureActors` si udržiava mapovanie z `PacketIngestorAndLoadBalancer` actorov na ich príslušných `Capture` actorov. Po prijatí správy `ProcessRawPacketBatchRequest` vyhľadá záznam v `captureActors` pre odosielateľa tejto správy. Ak pre neho našiel existujúceho `Capture` actora, prepošle mu túto správu. Ak pre daného `PacketIngestorAndLoadBalancer` actora neexistuje vytvorený `Capture` actor, dočasne si uloží túto správu (a všetky budúce správy, pokiaľ nie je vytvorený prislúchajúci `Capture` actor) a vyžiada si od neho (od `PacketIngestorAndLoadBalancer` actora) správou `CaptureInfoRequest` informácie o jeho aktuálne otvorenom zdroji paketov. Po prijatí odpovede typu `CaptureInfo`, `CapturesController` actor vytvorí príslušného `Capture` actora s odovzdanými prijatými informáciami o zdroji a prepošle mu všetky `ProcessRawPacketBatchRequest` správy, ktoré mal pre neho dočasne uložené.



Obr. 4.4: Triedny diagram `CapturesControllerActor`

Po prijatí správy `ProcessRawPacketBatchRequest` s nastaveným príznakom `FinalizeCapture`,

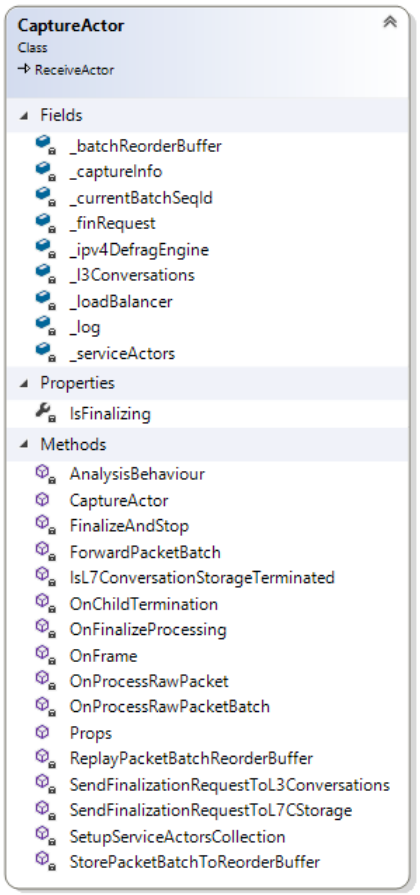
teda požiadavky na ukončenie spracovávania, táto správa nie je ďalej zaslaná príslušnému `Capture` actorovi, ale je mu namiesto nej zaslaná správa `FinalizeProcessingRequest`.

Capture

`Capture` actor (diagram na obrázku 4.5) pri svojom vytváraní preberá objekt `CaptureInfo`, ktorý špecifikuje zdroj paketov, ktorého obsah bude ďalej spracovávať (názov PCAP súboru). Po jeho spustení vytvorí ďalej `L7ConversationStorage` actora slúžiaceho na perzistentné ukladanie zrekonštruovaných `L7Conversation` objektov.

Pri prijatí správy `ProcessRawPacketBatchRequest` sú pomocou metódy `ForwardPacketBatch` vyextrahované jej jednotlivé `ProcessRawPacketRequest` objekty, ktoré sú ďalej zaslané samotnému `Capture` actorovi. Zároveň je príslušnému `PacketIngestorAndLoadBalancer` actorovi zaslaná správa `ProcessRawPacketBatchReponse`, ktorá potvrdzuje prijatie dávky.

Dávky nespracovaných paketov prijatých v správach `ProcessRawPacketBatchRequest` môžu byť však prijaté v inom poradí, rozdielnom od toho, v akom boli pôvodne odoslané. K tomuto môže dôjsť z dôvodu paralelného prenosu viacerých `ProcessRawPacketBatchRequest` správ súbežne actorom `PacketIngestorAndLoadBalancer`. Z tohto dôvodu sú tieto správy opatrené sekvenčným číslom, určujúcim ich korektné poradie. `Capture` actor si udržiava informáciu o očakávanom sekvenčnom čísle dávky a v prípade prijatia dávky s iným (vyšším) sekvenčným číslom než očakávaným, uloží si ju zoraďovacieho zásobníku. V ňom ostane, kým nie sú prijaté všetky jej predchádzajúce dávky. Zoradené dávky sú ďalej spracované metódou `ForwardPacketBatch`.



Obr. 4.5: Triedny diagram `CaptureActor`

Jednotlivé nespracované pakety `ProcessRawPacketRequest` sú prijaté metódou `OnProcessRawPacket`, ktorá ich zparsuje s využitím knižnice PacketDotNet a na základe zparsovaných paketov vytvorí príslušné inštancie triedy `Frame`. Tieto objekty sú ďalej zaslané samotnému `Capture` actorovi.

Trieda `Frame` implementuje rozhranie `IFrame`, ktoré popisuje zparsovaný paket, s ktorým ďalej pracujú zvyšní actori a reassembleri. Medzi vlastnosťami rozhrania `IFrame` sa nachádzajú okrem iných aj `L3ConversationKey` a `L4ConversationKey`, ktoré sú kľúčmi jednotlivých L3 a L4 konverzácií. `L3ConversationKey` je vypočítaný zo zoradeného páru zdrojovej a cieľovej IP adresy. `L4ConversationKey` je vypočítaný zo zoradeného páru zdrojového a cieľového portu a IP protokolu paketu. Z oboch kľúčov je možné vypočítať ich hash, vďaka čomu ich je možné využiť ako kľúče v slovníkoch.

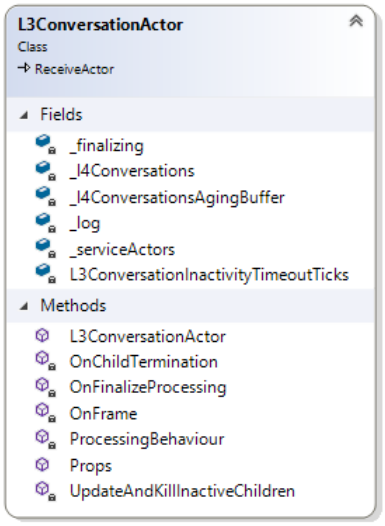
Spracované pakety `IFrame` sú prijaté metódou `OnFrame`. `CaptureActor` si udržuje mapovanie kľúčov L3 konverzácií na vytvorených `L3Conversation` actorov pomocou slovníka

`l3Conversations`. `OnFrame` tak pre daný paket na základe jeho kľúča L3 konverzácie určí, pre ktorého `L3 Conversation` actora by mal byť určený tento paket, ktorému ho následne pošle. Ak sa v `l3Conversations` nenájde záznam pre danú L3 konverzáciu, vytvorí sa pre ňu nový `L3 Conversation` actor a odkaz na neho sa pridá ako nový záznam v `l3Conversations` pre daný kľúč L3 konverzácie.

Po prijatí požiadavky na ukončenie spracovávania správou `FinalizeProcessingRequest`, `Capture` actor prepošle túto požiadavku všetkým `L3Conversation` actorom uloženým v `l3Conversations`. Počas svojej existencie zároveň monitoruje ukončenie svojich potomkov a tak keď je ukončený niektorý z jeho `L3Conversation` actorov, dôjde k jeho odstráneniu z `l3Conversations`. Keď je ukončený posledný `L3Conversation` actor, zašle príslušnému `PacketIngestorAndLoadBalancer` actorovi správu `ProcessRawPacketBatchReponse` indikujúcu úspešné dokončenie spracovávania. Po odoslaní tejto správy sa sám ukončí.

L3 Conversation

`L3Conversation` actor (diagram na obrázku 4.6) reprezentuje unikátnu L3 konverzáciu, ktorú ďalej rozdeľuje na L4 konverzácie. Podobne ako `Capture` actor, `L3Conversation` actor si udržuje slovník `l4Conversations` s mapovaním kľúčov L4 konverzácií na `L4Conversation` actorov.



Obr. 4.6: Triedny diagram `L3ConversationActor`

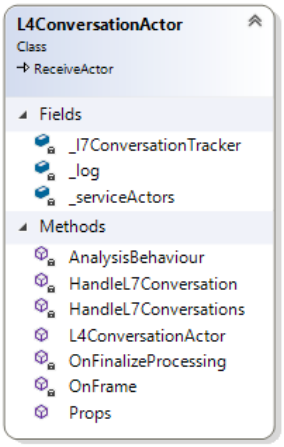
Prijaté správy `IFrame` od `Capture` actora sú spracované metódou `OnFrame`, ktorá ich prepošle príslušnému `L4Conversation` actorovi na základe kľúča L4 konverzácie prijatého paketu, ktorý sa použije pri získaní záznamu zo slovníku `l4Conversations`. Ak sa v `l4Conversations` nenájde záznam pre danú L4 konverzáciu, vytvorí sa pre ňu nový `L4 Conversation` actor a odkaz na neho sa pridá ako nový záznam v `l4Conversations` pre daný kľúč L4 konverzácie. Konštruktoru vytváraného `L4Conversation` actora sú odovzdané informácie o koncových bodoch komunikácie (zdrojová a cieľová IP adresa a port spracovávaného paketu) a IP protokol.

Po prijatí požiadavky na ukončenie spracovávania správou `FinalizeProcessingRequest`, `L3Conversation` actor, rovnako ako `Capture` actor prepošle túto požiadavku všetkým `L4Conversation` actorom v `l4Conversations`. Počas svojej existencie zároveň tiež monitoruje ukončenie svojich potomkov a tak keď je ukončený niektorý z jeho `L4Conversation` actorov, dôjde k jeho odstráneniu z `l4Conversations`. Keď je ukončený posledný `L4Conversation` actor, ukončí sa aj samotný `L3Conversation` actor.

`L3Conversation` actor si zároveň popri slovníku `l4Conversations` udržuje aj tzv. *aging buffer*, v ktorom si ukladá časové známky posledných zaznamenaných aktivít jednotlivých L4 konverzácií. Pomocou neho je schopný predčasne ukončiť neaktívne L4 konverzácie, pre ktoré je časový rozdiel medzi ich poslednou zaznamenanou aktivitou a časovou známkou aktuálneho paketu (aktuálne spracovávaného `IFrame` objektu) väčší ako definovaná hodnota pre timeout L7 konverzácií.

L4 Conversation

`L4Conversation` actor (diagram na obrázku 4.7) reprezentuje unikátnu L4 konverzáciu, ktorej pakety reassembluje s využitím reassembleru vytvoreného samostatne pre každú L4 konverzáciu. Objekt reassembleru je vytvorený pomocou továrne `L7ConversationTrackerFactory`, ktorá preberá koncové body a IP protokol danej L4 konverzácie. Z nich vytvorí konkrétny UDP alebo TCP reassembler, ktorý ďalej využije informácie o koncových bodoch na určenie relatívnych smerov budúcich paketov L4 konverzácie. Vytvorený reassembler je uložený do inštančnej premenej `l7ConversationTracker`.



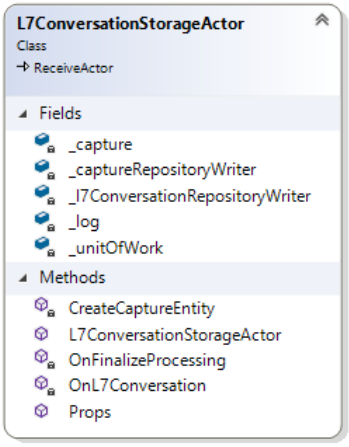
Obr. 4.7: Triedny diagram `L4ConversationActor`

Prijaté správy `IFrame` sú spracovávané metódou `OnFrame`, ktorá ich odovzdá ďalej ako parameter metóde `ProcessFrame` reassembleru `l7ConversationTracker`. V metóde `ProcessFrame` je tak spracovaný daný paket reassemblerom a pri prípadnom úspešnom zrekonštruovaní (prípadne viacerých) L7 konverzácií, sú tieto konverzácie vrátené metóde `OnFrame`, ktorá ich zašle actorovi `L7ConversationStorage` na uloženie do perzistentného úložiska.

Po prijatí požiadavky na ukončenie spracovávania správou `FinalizeProcessingRequest`, `L4Conversation` actor zavolá metódu `Complete` reassembleru `l7ConversationTracker`, ktorou ho inštruuje, aby dokončil reassemblovací proces. Zrekonštruované L7 konverzácie sú opäť zaslané actorovi `L7ConversationStorage`. Po odoslaní prípadných L7 konverzácií sa ukončí.

L7 Conversation Storage

`L7ConversationStorage` actor (diagram na obrázku 4.8) slúži na perzistentné ukladanie zrekonštruovaných L7 konverzácií z daného zdroja paketov. Na implementáciu perzistentného ukladania objektov je využitá knižnica UnitOfWork[11], ktorá implementuje návrhový vzor UnitOfWork a Repository. Táto knižnica tak poskytuje jednotný spôsob prístupu k poskytovateľom databázového prístupu ako Cassadra, Arrango a Entity Framework. V základnom nastavení `L7ConversationStorage` používa ukladanie perzistentných objektov len do pamäte. Pre testovacie účely je možné zakázať perzistentné ukladanie. Prijaté `L7Conversation` správy sú pracované metódou `OnL7Conversation`, ktorá ich uloží do príslušného repozitáru.



Obr. 4.8: Triedny diagram `L7ConversationStorageActor`

[11] UnitOfWork, https://github.com/nesfit/UnitOfWork

4.4.2 Implementácia reassemblingu

UDP aj TCP reassembler je implementovaný v samostatnom projekte NTPAC.Reassembling, v triedach `UdpConversationTracker` a `TcpConversationTracker`. Obe tieto triedy sú odvodené od spoločnej abstraktnej nadtriedy `L7ConversationTrackerBase`, ktorá definuje ďalej abstraktnú metódu `ProcessFrame` určenú na spracovanie nasledujúceho `IFrame` objektu a metódu `Complete` na dokončenie reassemblovania. Obe tieto metódy vracajú zrekonštruované L7 konverzácie (ak sa nejaké podarilo v danom kroku zrekonštruovať). Konštruktor triedy `L7ConversationTrackerBase` preberá informácie o koncových bodoch L4 konverzácie, ktoré sú využité metódou `GetFrameFlowDirection`, ktorá pre daný `IFrame` určí jeho relatívny smer v L4 konverzácii. Obidva reassemblery pri svojom fungovaní využívajú sadu doplňujúcich modelov:

`ReassembledL7Pdu` implementujúci rozhranie `IL7Pdu`, je využitý pri postupnom zostavovaní L7 PDU. Metódou `AddFrame` je možné tomuto L7 PDU postupne pridávať pakety typu `IFrame`. Výsledný obsah L7 dát sa získa konkatenáciou L7 dát jednotlivých `IFrame` objektov.

`L7Flow` implementujúci rozhranie `IL7Flow` predstavuje jednosmerný L7 tok. Objekt tejto triedy obsahuje kolekciu `IL7Pdu` objektov predstavujúcich jednotlivé L7 PDU a kolekciu `IFrame` objektov, v ktorej ukladá pakety, ktoré nenesú v sebe žiadne L7 dáta a tak nie sú súčasťou žiadneho L7 PDU (napríklad pakety TCP handshaku). Ďalej obsahuje svoj identifikátor, využitý pri párovaní s tokom opačného smeru a pravdivostnú hodnotu indikujúcu, či je daný tok spárovaný s tokom opačného smeru.

`ReassembledL7Conversation` , ktorý je podtriedou `L7Conversation`. Táto implementácia L7 konverzácie je vytváraná na základe spojenia dvoch spárovaných jednosmerných L7 tokov (`L7Flow` objektov), kde obidva obsahujú kolekciu L7 PDU vo svojom smere. `ReassembledL7Conversation` tak na rozdiel od `L7Conversation` používa dve kolekcie L7 PDU, jednu pre každý smer a jednotnú kolekciu L7 PDU z oboch smerov buduje na vyžiadanie pomocou zjednotenia týchto dvoch kolekcií so zoradením jednotlivých L7 PDU v čase.

UDP Conversation Tracker

UDP reassembler je implementovaný triedou `UdpConversationTracker`. Udržuje si dve inštancie `L7Flow` objektov, každú pre jeden tok komunikácie.

Po prijatí `IFrame` objektu metódou `ProcessFrame` vypočíta časový rozdiel medzi poslednou aktivitou oboch `L7Flow` objektov a časovou známkou prijatého paketu a ak je väčší ako definovaná hodnota – 10 minút, znamená to že aktuálne spracovávaný paket patri do novej L7 konverzácie a tak predchádzajúcu L7 konverzáciu môžeme prehlásiť za ukončenú. Z vytvorených `L7Flow` objektov je tak vytvorený jeden `ReassembledL7Conversation` objekt a obidva `L7Flow` objekty sú následne vynulované (dealokované), čím sa sa pripraví prostredie pre tvorbu novej L7 konverzácie.

`ProcessFrame` ďalej pokračuje pridaním daného `IFrame` objektu do aktuálnej L7 konverzácie. Za pomoci metódy `GetFrameFlowDirection` určí relatívny smer daného paketu a získa (prípadne vytvorí ak nie je vytvorený) príslušný `L7Flow`. Následne vytvorí novú inštanciu `ReassembledL7Pdu` obsahujúcu prijatý `IFrame` objekt, ktorú ďalej pridá do príslušného `L7Flow` objektu. Návratovou hodnotou tejto metódy je prípadná ukončená L7 konverzácia – inštancia `ReassembledL7Conversation`. Metódou `Complete` je ukončená aktuálna L7

konverzácia. Vytvorená inštancia `ReassembledL7Conversation` je následne vrátená z tejto metódy.

TCP Conversation Tracker

TCP reassembler je implementovaný triedou `TcpConversationTracker`. Na rozdiel od UDP reassembleru nevytvára priamo objekty `L7Flow`, ale využíva dvojicu reassemblerov jednosmerných TCP tokov `TcpFlowReassembler`, ktoré vytvárajú ďalej zrekonštruované toky `L7Flow`. Tie sú vo finálnej fáze (metódou `Complete`) spárované a z jednotlivých dvojíc tokov sú zostavené L7 konverzácie `ReassembledL7Conversation`.

Metóda `ProcessFrame` objektu `TcpFlowReassembler` uloží odovzdaný `IFrame` do kolekcie `ReassemblingCollection`, ktorá v sebe zoraďuje jednotlivé `IFrame` objekty podľa ich TCP sekvenčných čísel. Tá ich interne ukladá do dvojitého linked listu a pri procese hľadania vhodného umiestnia vkladaného `IFrame` objektu využíva posledný vložený `IFrame`, od ktorého sa postupne ďalej posúva položkami linked listu až do momentu, kedy nájde vhodné umiestnenie. Týmto je minimalizovaný počet nutných priechodov uložených položiek.

Pri zavolaní metódy `Complete` objektu `TcpFlowReassembler` sú zo zoradených `IFrame` objektov kolekcie `ReassemblingCollection` postupne vytvárané L7 PDU `ReassembledL7Pdu` a z nich jednosmerné L7 toky `L7Flow` spôsobom, aký bol navrhnutý v podkapitole 3.5. Vytvorené `L7Flow` sú uložené do zoznamu `CompletedFlows`.

Metóda `ProcessFrame` objektu `TcpConversationTracker` pre daný `IFrame` vyberie na základe jeho relatívneho smeru jeden z dvojice `TcpFlowReassembler`-ov (prípadne ho vytvorí, ak pre zvolený smer nie je ešte vytvorený), ktorému ďalej odovzdá metódou `ProcessFrame` daný `IFrame` objekt.

Po odovzdaní všetkých paketov `IFrame` danej L4 konverzácie je zavolaná metóda `Complete` objektu `TcpConversationTracker`, ktorá zavolá ďalej metódy `Complete` oboch `TcpFlowReassemer`-ov a následne z ich `CompletedFlows` vytvorí dvojice objektov `L7Flow`, ktoré patria do jednej L7 konverzácie (podľa zhodného identifikátoru toku alebo prieniku v čase). Z každej nájdenej dvojice je vytvorená finálna L7 konverzácia, inštancia triedy `ReassembledL7Conversation`. Z tokov, ku ktorým nebol nájdený príslušný tok opačného smeru, sú vytvorené jednosmerné L7 konverzácie.

4.5 Použitie a testovanie

Výsledný nástroj NTPAC je tvorený dvoma programami NTPAC.LoadBalancerCli a NTPAC.Reassembler a jedným doplňujúcim programom Lighthouse.NetCoreApp, ktorý slúži ako seed uzol Akka.NET clustera. Pre zjednotenie prekladu a spustenia sú jednotlivé programy umiestnené v samostatných Docker kontajneroch.

NTPAC.Reassembler a Lighthouse.NetCoreApp sú určené na spustenie ako služby, tj. nemajú dopredu ohraničenú dobu vykonávania. NTPAC.LoadBalancerCli sa spúšťa jedenkrát pre daný PCAP súbor definovaný vstupným argumentom príkazového riadka. V prílohe B sú vypísané nápovedy jednotlivých programov, popisujúce možné argumenty príkazového riadka (k vypísaniu nápovedy dôjde pri spustení s argumentom `--help`).

Na priloženom DVD sa nachádza video zachytávajúce priebeh spracovávania vstupného PCAP súboru na jednom počítači. V prvom kroku je spustený seed uzol Lighthouse.NetCoreApp, následne dve inštancie NTPAC.Reassembler uzlov a nakoniec jedna inštancia NTPAC.LoadBalancerCli s argumentom príkazového riadku špecifikujúcim cestu

PCAP súboru na analýzu. Počas analýzy sú na *Reassembler* uzloch priebežne vypisované informácie o jednotlivých zrekonštruovaných L7 konverzáciách.

Testovanie implementácie prebiehalo formou unit testov, ktoré boli primárne zamerané na korektný reassembling UDP a TCP konverzácií. Tieto unit testy používali vybrané testovacie PCAP súbory z priečinku *NTPAC/TestingData*, ktoré obsahovali vždy jednu samostatnú L4 konverzáciu. Každý test z príslušného testovacieho PCAP súboru prečítal obsiahnuté pakety ako `IFrame` objekty, ktoré spracoval metódou `ProcessFrame` vytvoreného `UdpConversationTracker`-u alebo `TcpConversationTracker`-u, na čo získal jednotlivé zrekonštruované L7 konverzácie `L7Conversation` zavolaním metódy `Complete`. Nakoniec bolo overené, že bol zrekonštruovaný očakávaný počet L7 konverzácií a že aj počet a obsah ich L7 PDU odpovedá skutočnosti.

Kapitola 5

Model výpočtového clusteru a výkonnostné testovanie

Aby bolo možné otestovať vzájomnú komunikáciu a funkčnosť podsystémov naimplementovaného distribuovaného systému NTPAC so súčasným priblížením sa reálnemu použitiu, bol zostavený malý výpočtový cluster. Táto kapitola sa venuje jednodoskovým počítačom, ich využiteľnosti pri stavbe výpočtových clusterov a popisu jedného takého zloženého z počítačov Banana Pi R2.

Aj keď sa svojím výkonom jednodoskové počítače nepribližujú moderným serverom a osobným počítačom, cluster zložený z nich môže stále slúžiť na predvedenie a testovanie funkčnosti distribuovaného systému pri zachovaní nízkej celkovej ceny. Pri následnom praktickom využití systému bude môcť byť využitý reálny výpočtový cluster.

5.1 Jednodoskové počítače

Jednodoskové počítače sú malé, jednoduché počítače ktoré obsahujú všetky svoje komponenty ako procesor, pamäť a IO na jednom plošnom spoji. Vyznačujú sa nízkou cenou a spotrebou elektrickej energie. Ich uplatnenie je preto široké, od výukových účelov, cez vstavané zariadenia, až po jednoduché herné konzoly. V dnešnej dobe je najznámejším a najrozšírenejším jednodoskovým počítačom Raspberry Pi vyvíjaný spoločnosťou Raspberry Pi Foundation. Od začiatku predaja v roku 2012 do júla 2017 bolo predaných takmer 15 miliónov kusov[1].

5.2 Banana Pi R2 cluster

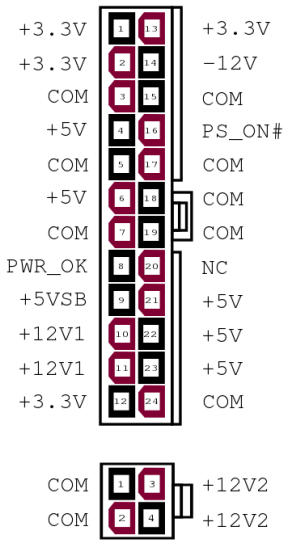
Náš cluster pozostáva z celkovo siedmych jednodoskových počítačov Banana Pi R2 vyvíjaných spoločnosťou SinoVoip. Ich hlavné charakteristiky, podstatné pre naše účely, sú štvorjadrový procesor ARM Cortex-A7 o maximálnom takte 1.3GHz, 2GB DDR3 SDRAM, 8GB eMMC flash pamäť a 5 portový 1Gb ethernet switch (s dvoma sieťovými rozhraniami)[2]. Vďaka tomu poskytujú väčší dostupný výkon a rýchlosť siete ako už spomínané Raspberry

[1] Eben Upton, "Raspberry Pi founder Eben Upton talks sales numbers, proudest moments, community projects, and Raspberry Pi 4", 2017, https://betanews.com/2017/07/19/raspberry-pi-eben-upton-qa/

[2] https://bananapi.gitbooks.io/banana-pi-bpi-r2-open-source-smart-router/content

Pi, ktorého súčasný model 3 disponuje štvorjadrovým procesorom ARM Cortex-A53 o maximálnom takte 1.2GHz, 1GB LPDDR2 a jedným 100Mb ethernetovým rozhraním[3].

Celý cluster je možné napájať 300W ATX12V zdrojom napojeným na príslušné konektory Molex 39-01-2200 (20 pinový konektor) a Molex 39-01-2040 (4 pinový konektor), ktorých pinout je možné vidieť na obrázku 5.1. Aby sa pri práci zvýšila úroveň komfortu a kontroly nad celým systémom, bola doň zakomponovaná možnosť diaľkovej aktivácie napájacieho zdroja a ovládania prívodu napájania do jednotlivých dosiek. To bolo docielené pridaním 8 kanálovej reléovej dosky a mikrokontroléru ESP32, ktorý ju ovláda. ATX zdroje je možné naplno aktivovať privedením signálu `PS_ON#` nachádzajúcom sa na 20 pinovom konektore[4] na zem. Tento signál je preto spojený so zemou cez relé na relé doske (v režime Normal Open), vďaka čomu je možné jeho zopnutím aktivovať celý napájací zdroj. Cez zvyšných sedem relé je pripojených (taktiež v režime Normal Open) napájacie napätie 12V k jednotlivým doskám. Na obrázku 5.2 je zobrazená bloková schéma jednotlivých komponentov clustera.



Obr. 5.1: Pinout 20 a 4 pinových ATX Molex konektorov

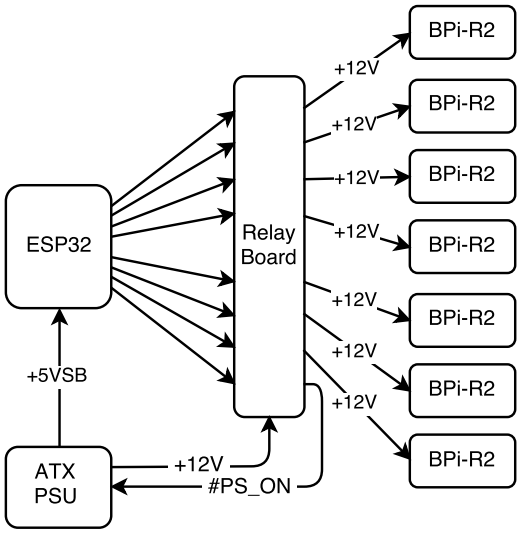
Všetky popísané komponenty, teda Banana Pi R2 dosky, mikrokontrolér ESP32 a relé doska, sú uložené vo zvislej polohe spojené plastovými dielmi, ktoré boli špeciálne navrhnuté a vytlačené na 3D tlačiarni. Tieto diely slúžia zároveň aj ako podstava celého clustera a pomáhajú aj pri vedení kabeláže. Na obrázku 5.3 je možné vidieť finálnu podobu zostaveného systému.

ATX zdroje poskytujú na vodiči `+5VSB` neustálych 5V, aj keď nie sú v aktívnom stave. Nimi je napájaný mikrokontrolér ESP32 od spoločnosti Espressif. Ten disponuje 160 MHz 32 bitovým mikroprocesorom Tensillizca Xtensa LX6 s 520 KB SRAM, 2 MB flash pamäte a WLAN sieťovou konektivitou[5]. Tieto hardwarové špecifiká umožnili na mikrokontroléri implementáciu webového serveru spolu s webovou aplikáciou určenej na manipuláciu s jed-

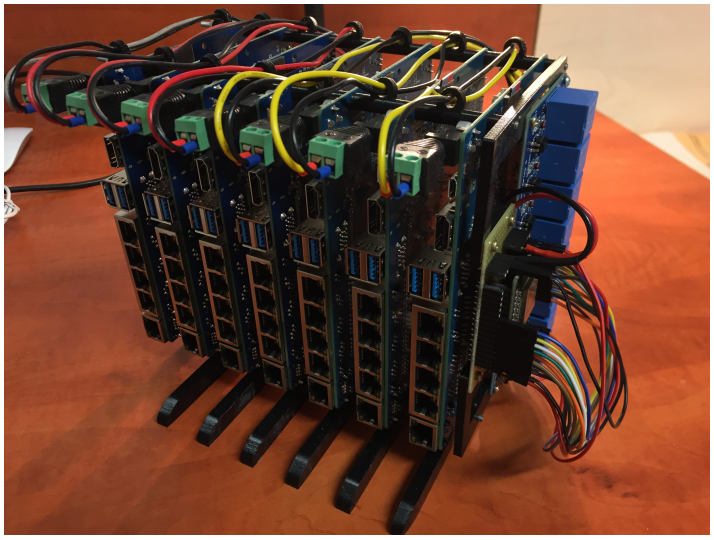
[3]The MagPi MagazineThe MagPi Magazine, https://www.raspberrypi.org/magpi/raspberry-pi-3-specs-benchmarks

[4]Vodič typicky označený zelenou farbou.

[5]http://espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf



Obr. 5.2: Bloková schéma clustru



Obr. 5.3: Podoba zostaveného clustera

notlivými relé pripojenej relé dosky. Pri vývoji firmvéru bola použitá open source platforma PlatformIO[6], určená na vývoj IoT zariadení. Tá okrem iného umožňuje OTA (Over the Air) nahrávanie firmvéru, čo zjednodušilo a urýchlilo proces jeho vývoja. Na úrovni zdrojového kódu bola použitá Arduino ESP32 knižnica[7], emulujúca programové rozhranie Arduina, čo tiež zjednodušilo vývoj. Zdrojové súbory firmwéru sa nachádzajú v priložených súboroch v priečinku CPC. V hlavnom zdrojovom súbore *src/cpc.cpp* je možné pomocou globálnych premenných `wlanSettings`, `appUser`, `appPassword`, `appPort` a `relays` nakonfigurovať prístup do (prípadne viacerých) WLAN sietí, meno, heslo a port webovej aplikácie a mapovanie jednotlivých relé na piny ESP32. Chod firmwéru je možné popísať týmito krokmi:

1. Inicializácia IO portov pre relé, sériového portu a súborového systému.

[6] https://platformio.org
[7] arduino-esp32, https://github.com/espressif/arduino-esp32

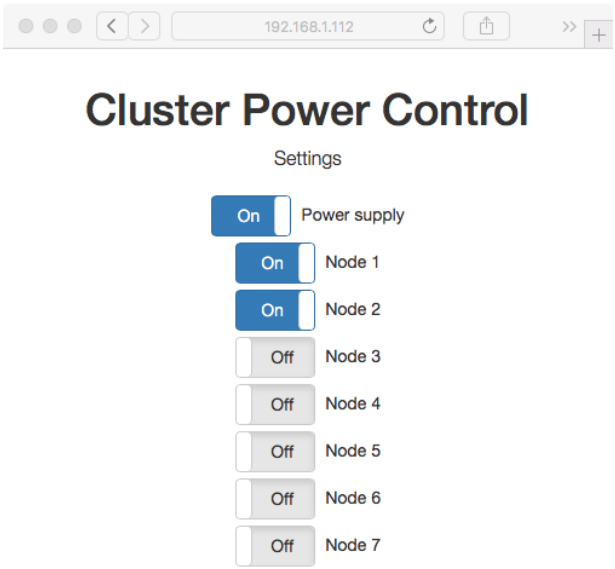
2. Pripojenie k niektorej zo známych WLAN sietí a automatická IP konfigurácia pomocou protokolu DHCP. Pridelená IP adresa je vypísaná na sériový port. V prípade ak nedošlo k žiadnemu úspešnému pripojeniu, mikrokontrolér je po piatich sekundách reštartovaný.

3. Inicializácia OTA a web serveru.

4. Rozsvietenie vstavanej modrej LED, indikujúc úspešnú inicializáciu a dostupnosť webovej aplikácie.

5. Nekonečný cyklus obsluhy OTA a webových požiadaviek.

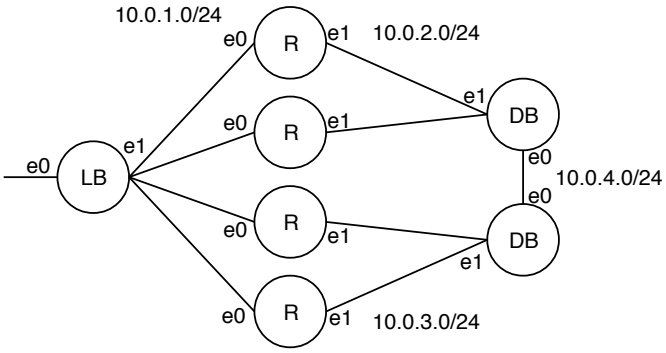
Webová aplikácia, ktorej ukážku je možné vidieť na obrázku 5.4, využíva HTTP API, pomocou ktorého je možné zasielať mikrokontroléru príkazy na zapnutie alebo vypnutie príslušných relé. Na zaistenie základného zabezpečenia pred neautorizovaným prístupom a manipuláciou bola použitá HTTP Basic Auth vyžadujúca zadanie mena a hesla.



Obr. 5.4: Ukážka webovej aplikácie na ESP32

Topológia a konfigurácia

Každý uzol (Banana Pi R2 doska) vlastní dve sieťové rozhrania, *e0* a *e1*. K rozhraniu *e1* je interne pripojený štvorportový switch. Pre napodobenie reálneho použitia bola zvolená zostava zložená z jedného *LoadBalancer* uzlu, štyroch *Reassembler* uzlov a dvoch databázových uzlov. Jednotlivé uzly sú zapojené v topológii znázornenej na obrázku 5.5. Cluster interne využíva adresný priestor 10.0.0.0/21 s využitím statického smerovania. Tento adresný priestor je skrytý za NATom na *LoadBalancer* uzli, ktorý slúži ako vstupný bod do clustera a východisková cesta (default route) pre zvyšné uzly. Jednotlivé uzly využívajú operačný systém Linux s distribúciou Ubuntu.



Obr. 5.5: Topológia clustra

5.3 Výkonnostné testovanie

Nástroj NTPAC je postavený na knižnici Akka.NET, ktorá na implementáciu sieťovej komunikácie medzi uzlami využíva knižnicu dotnetty, v dobe písania tejto práce vo verzii 0.4.6. Pri vývoji sa však ukázalo, že táto verzia DotNetty obsahuje doposiaľ neopravený bug postihujúci operačný systém Linux, spôsobujúci stratu synchronizácie dátových rámcov medzi koncami komunikácie pri zasielaní správ. Zostavený Banana Pi cluster tak nemohol byť do odovzdania tejto práce využitý na testovanie nástroja NTPAC. Namiesto neho muselo testovanie prebehnúť na sade osobných počítačov.

Na testovanie bolo využitých 5 osobných počítačov, každý s procesorom i5-3570K 3.4 GHz, 16 GB RAM, 1 Gbps NIC[8] a operačným systémom Windows 10, prepojených spolu pomocou 1Gbs switch. Ako testovacie dáta slúžili PCAP súbory *isa-http.pcap* (780 MB, 1332293 paketov), *sec6net.pcap* (5.16 GB, 4551241 paketov) a *all.pcap* (5.32 GB, 4619556 paketov). Na každom počítači bol nasadený vždy maximálne jeden uzol. Testy prebiehali s využitím jedného *LoadBalancer* uzlu s postupným pridávaním *Reassembler* uzlov. V tabuľkách 5.1, 5.2 a 5.3 je vidieť doby strávené spracovávaním súborov *isa-http.pcap*, *sec6net.pcap* a *all.pcap*, spolu s prepočítanou rýchlosťou spracovávania v Mb/s vzhľadom na veľkosť vstupného PCAP súboru. Každé meranie prebehlo trikrát a výsledný čas testu je reprezentovaný priemernou hodnotou výsledkov týchto meraní. Zrekonštruované L7 konverzácie boli zahadzované (neboli ukladané do pamäte).

Reassembler uzlov	Čas [s]	Rýchlosť [Mb/s]
1	20	315
2	18	350
3	15.6	404
4	15.5	406

Tabuľka 5.1: Výkonnostné meranie spracovávania testovacieho PCAP súboru *isa-http.pcap* s postupým pridávaním *Reassembler* uzlov

[8] Network Interface Card

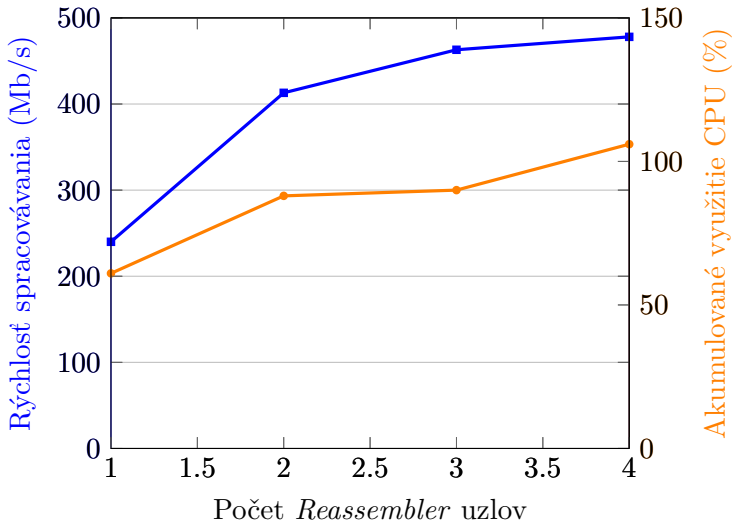
Reassembler uzlov	Čas [s]	Rýchlosť [Mb/s]
1	122	338
2	108	382
3	98	421
4	98.5	419

Tabuľka 5.2: Výkonnostné meranie spracovávania testovacieho PCAP súboru *sec6net.pcap* s postupým pridávaním *Reassembler* uzlov

Reassembler uzlov	Čas [s]	Rýchlosť [Mb/s]
1	176	240
2	103	413
3	92	463
4	89	478

Tabuľka 5.3: Výkonnostné meranie spracovávania testovacieho PCAP súboru *all.pcap* s postupým pridávaním *Reassembler* uzlov

Z nameraných výsledkov vidíme, že rýchlosť spracovávania súborov *isa-http.pcap* a *sec6net.pcap* sa na počítačoch ustálila na približne 420 Mb/s s použitím troch *Reassembler* uzlov. Pri tejto rýchlosti bola linka na *LoadBalancer* uzle využitá na približne 500 Mb/s, Na celkový výkon (rýchlosť spracovávania) nástroja mala negatívny vplyv nevyváženosť rozloženia práce, kedy bol väčšinou vyťažený spracovávaním len jeden *Reassembler* uzol (približne 70% vyťaženie CPU) a zvyšné pracovali len na zlomok svojho plného potenciálu (približne 20% vyťaženie CPU). Toto bolo spôsobené tým, že väčšina paketov v oboch testovacích PCAP súboroch patrila do jedinej L4 konverzácie. Rýchlosť spracovávania súboru *all.pcap*, ktorého pakety boli rovnomernejšie rozdelené do L4 konverzácií sa pohybovala okolo 478 Mb/s. V tabuľke 5.4 je vidieť priemerné vyťaženie CPU a úroveň zaplnenia pamäte RAM jednotlivých (zapojených) uzlov pri postupnom pridávaní *Reassembler* uzlov. Meranie týchto hodnôt bolo vykonané nástrojom Performance Monitor s čítačmi Processor/Processor Time a Memory/Commited Bytes. Programovací jazyk C# využíva Garbage Collector na automatickú správu alokovaných objektov a tak namerané hodnoty úrovní zaplnenia pamäte RAM sú ovplyvnené doposiaľ neuvoľnenými objektami. Skratka LB v hlavičke tabuľky značí *LoadBalacer* uzol a skratka Rn jeden z *Reassembler* uzlov. V grafe 5.6 je ďalej vidno závislosť medzi rýchlosťou spracovávania a akumulovaným využitím CPU *Reassembler* uzlov (prepočítaného na výkon jedného CPU) pre daný počet *Reassembler* uzlov.



Obr. 5.6: Závislosť medzi rýchlosťou spracovávania a akumulovaným využitím CPU *Reassembler* uzlov (prepočítaného na výkon jedného CPU) pre daný počet *Reassembler* uzlov

n	LB CPU [%]	LB Mem [GB]	R1 CPU [%]	R1 Mem [GB]	R2 CPU [%]	R2 Mem [GB]	R3 CPU [%]	R3 Mem [GB]	R4 CPU [%]	R4 Mem [GB]
1	17	5.5	61	7.4	-	-	-	-	-	-
2	27	5.4	40	6.5	48	7	-	-	-	-
3	25	5.5	30	6.2	30	4,3	30	6.5	-	-
4	30	5.5	26	5.9	26	5.9	26	3.8	28	6.3

Tabuľka 5.4: Zaznamenanie priemerných úrovní vyťaženia CPU a zaplnenia pamäte RAM jednotlivých (zapojených) uzlov pri spracovávaní testovacieho PCAP súboru *all.pcap*, pri postupnom pridávaní *Reassembler* uzlov

Kapitola 6

Záver

Táto práca sa venovala možnostiam rekonštrukcie zachytenej komunikácie v distribuovanom prostredí. Tento prístup si berie za cieľ zvýšenie objemu dát, ktoré je možné analyzovať v danom čase pri forenznej analýze sieťovej komunikácie. Bol navrhnutý a implementovaný distribuovaný systém postavený na Actor modeli určený na rekonštrukciu L7 konverzácií z paketov vstupných PCAP súborov.

Kapitola 2 popisovala problematiku rekonštrukcie zachytenej sieťovej komunikácie. V prvej časti popísala TCP/IP model, jeho jednotlivé vrstvy, radenie paketov do konverzácií na úrovniach týchto vrstiev a spôsob, akým môžeme reprezentovať prenesené aplikačné dáta. Ďalej bližšie popísala IP protokoly UDP a TCP, spôsoby, akými prenášajú aplikačné dáta a nakoniec ako je možné z týchto aplikačných dát rekonštruovať ďalej jednotlivé aplikačné správy.

Kapitola 3 sa venovala definícii distribuovaných systémov spolu s modelmi súbežného vykonávania, konkrétne Actor modelu a Komunikujúcim sekvenčným procesom. Ďalej sa táto kapitola venovala architektúrnemu návrhu nástroja NTPAC za využitia Actor modelu, za ktorým nasledoval bližší návrh jeho podsystémov, uzlov *LoadBalancer* a *Reassembler* distribuovaného systému. Návrh *LoadBalancer* uzla popisoval hlavne metódy, akými distribuovať vstupné pakety jednotlivým *Reassembler* uzlom na spracovanie. Návrh *Reassembler* uzlu popisoval rozdelenie paketov do L3 a L4 konverzácií za pomoci stromovej štruktúry actorov a následne metódy UDP a TCP reassemblingu L7 konverzácií z paketov L4 konverzácií.

V kapitole 4 bola popísaná implementácia navrhnutých uzlov *LoadBalancer* a *Reassembler* v programovacom jazyku C# s využitím knižnice Akka.NET.

V poslednej kapitole 5 bol popísaný modelový cluster zostavený z jednodoskových počítačov Banana Pi R2, ktorý bol navrhnutý a zostavený na účely otestovania implementovaného nastroja NTPAC. Chyba v knižnici dotnetty, použitej knižnicou Akka.NET na implementáciu TCP komunikácie ale spôsobila neschopnosť vzájomnej komunikácie jednotlivých uzlov využívajúcich operačný systém Linux, ktorý využívali aj samotné počítače Banana Pi R2. Testovanie tak muselo prebehnúť na osobných počítačoch s operačným systémom Windows 10.

Cieľ tejto práce bol dosiahnutý. Navrhnutý a implementovaný nástroj NTPAC je schopný distribuovane zrekonštruovať L7 konverzácie z paketov vstupného PCAP súboru. Logickým pokračovaním tejto práce je implementácia uzlov určených na rekonštrukciu aplikačných správ zo zrekonštruovaných L7 konverzácií. Ďalej je priestor aj na výkonnostné vylepšenia a optimalizácie, ako napríklad implementácia detekcie neaktívnych L3 konverzácií. Táto práca bola vypracovaná v rámci projektov Tarzan a IGA.

Literatúra

[1] Akka.NET Documentation. 2017.
URL https://getakka.net/articles/intro/what-is-akka.html

[2] Davidoff, S.; Ham, J.: *Network Forensics: Tracking Hackers through Cyberspace*. Prentice Hall, 2012, ISBN 0132565102.

[3] Hewitt, C.; Bishop, P.; Steiger, R.: A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, Morgan Kaufmann Publishers Inc., 1973, s. 235–245.

[4] Hoare, C. A. R.: Communicating Sequential Processes. *Commun. ACM*, ročník 21, č. 8, Srpen 1978: s. 666–677, ISSN 0001-0782.

[5] Kshemkalyani, A. D.; Singhal, M.: *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2011, ISBN 0521189845.

[6] Matoušek, P.; Pluskal, J.; Ryšavý, O.; aj.: Advanced techniques for reconstruction of incomplete network data. In *International Conference on Digital Forensics and Cyber Crime*, Springer, 2015, s. 69–84.

[7] Palmer, G.: A Road Map for Digital Forensic Research. Technická zpráva, The Digital Forensic Research Conference, 2001.

[8] Pluskal, J.: *Framework for Captured Network Communication Processing*. Diplomová práce, Brno University of Technology, Faculty of Information Technology, 2014.
URL http://www.fit.vutbr.cz/study/DP/DP.php?id=16748

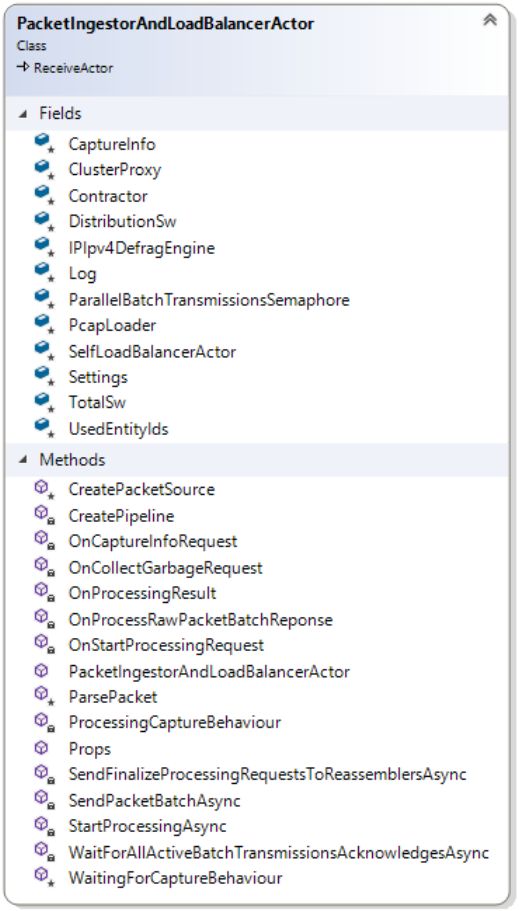
[9] Pluskal, J.: Netfox Detective 2.0 - Nástroj pro síťovou forenzní analýzu. Technická zpráva, Fakulta informačních technologií, Vysoké učení technické v Brně, 2017.

[10] Postel, J.: User Datagram Protocol. Technická Zpráva 768, Srpen 1980.
URL http://www.ietf.org/rfc/rfc768.txt

[11] Postel, J.: Transmission Control Protocol. Technická Zpráva 793, Září 1981, updated by RFCs 1122, 3168, 6093, 6528.
URL http://www.ietf.org/rfc/rfc793.txt

Príloha A

Doplňujúce triedne diagramy



Obr. A.1: Triedny diagram `PacketIngestorAndLoadBalancerActor`

Príloha B

Nápovedy k podprogramom nástroja NTPAC

```
$ ./Lighthouse.NetCoreApp --help
Lighthouse 1.0.0
Copyright (C) 2018 Lighthouse.NetCoreApp
  -h, --hostname    (Default: 127.0.0.1) IP address to listen on
                        and to be *reachable* at
  -p, --port        (Default: 7070) Port
  --help            Display this help screen.
  --version         Display version information.

$ ./NTPAC.LoadBalancerCli --help
NTPAC 1.0.0
Copyright (C) 2018 NTPAC.LoadBalancerCli
  --offline               (Default: false) Run in offline mode
                              outside of the cluster.
  --debug                 (Default: false) Enable debug logging
  --devnull-repository    (Default: false) Don't save any persisted objects
  -h, --hostname          (Default: 127.0.0.1) IP address to listen on
                              and to be *reachable* at
  -p, --port              (Default: 0) Port (0 - random)
  -s, --seednode          (Default: 127.0.0.1:7070) Hostname
                              and port of the cluster seed node
  --help                  Display this help screen.
  --version               Display version information.
  value pos. 0            Required. Pcap file path.

$ ./NTPAC.Reassembler --help
NTPAC 1.0.0
Copyright (C) 2018 NTPAC.Reassembler
  -h, --hostname          (Default: 127.0.0.1) IP address to listen on
                              and to be *reachable* at
  -p, --port              (Default: 0) Port (0 - random)
  --devnull-repository    (Default: false) Don't save any persisted objects
```

```
-s, --seednode          (Default: 127.0.0.1:7070) Hostname
                            and port of the cluster seed node
--help                  Display this help screen.
--version               Display version information.
```

Príloha C

Obsah DVD

Priložené DVD obsahuje:

1. Text diplomovej práce vo formáte PDF.
2. Zdrojové súbory diplomovej práce pre systém LaTeX.
3. Zdrojové súbory nástroja NTPAC.
4. Zdrojové súbory firmwéru CPC.
5. Sada testovacích PCAP súborov.
6. Demonštračné video zachytávajúce použitie nástroja.